Software Process Improvement Center

Technical Report

프로세스 모델링 언어와 자동화 환경 비교

SPIC-TR-20030728-1-PB-001

2003년 7월 28일

소프트웨어 프로세스 개선 센터

한국과학기술원(KAIST)

대전시 유성구 구성동 373-1

TEL : 042-869-8445

http://spic.kaist.ac.kr

☐ 본 연구는 대학IT연구센터(ITRC : Information Technology Research Center)로 지정되어 정보통신부로부터 지원 받고 있습니다.

프로세스 모델링 언어와 자동화 환경의 비교

- 과제 : 제1 세부과제
- 소속 : 한국과학기술원
- 책임자 : 배두환
- 작성연구원
  최경식, 윤경아, 전상욱, 김철호, 손명근, 고현민, 이선애

- 제출일 : 2003년 7월 28일

센터장 : 배두환 (인)

목 차

그림 목차

1. 개요

1987년 Osterweil의 논문[1]에서 “소프트웨어 프로세스(software process)도 소프트웨어”라는 주장이 제기되었다. 이는 소프트웨어 프로세스도 소프트웨어처럼 도메인에서 필요한 요구사항에 따라 분석 및 설계 되고, 구현될 수 있기 때문에 이 과정을 통해 소프트웨어 프로세스를 자동화(process automation)시킬 수 있다는 주장이었다. Osterweil의 의견에 따라 이와 관련한 많은 연구들이 진행되었고, 대체로 많은 연구가 소프트웨어 프로세스를 모델링 하고, 이를 자동화하는 환경을 제공하는데 집중되었다. 소프트웨어 프로세스 모델링 시 사용하는 언어를 프로세스 모델링 언어(Process Modeling Language: PML)이라 하고, PML을 통해 프로세스를 모델링하고 실행의 자동화를 지원해 주는 환경을 PSEE(Process centered Software Engineering Environment)라고 한다. 본 문서는 소프트웨어 프로세스 모델링과 자동화 환경에 대한 과거의 연구들을 제공하여, 새로운 개념과 형태의 소프트웨어 프로세스 자동화 환경을 제안하는데 기반이 되고자 하는 것을 목적으로 한다. 본 장에서는 프로세스 모델링과 PSEE에 대한 개요를 설명하고 2장부터 10장까지는 다양한 개념들을 제공하고 있는 선별된 연구들에 대해 설명한다.

1.1 프로세스 자동화의 필요성

PSEE는 개발지원 도구와 개발 프로세스를 가지고 소프트웨어를 개발하는 조직에 소속된 사람들을 통합하는 수단을 제공한다. 이의 필요성에 대해 설명하기 위한 간단한 사례를 들면 다음과 같다. 먼저, 소프트웨어 개발업무 중 중요한 안건의 처리 도중 중간 관계자가 자신이 처리할 일을 잊고 처리하지 않은 경우 업무 상 치명적인 결과를 초래할 수 있다. 또한 프로세스에 대한 여러 측면에서 측정(measurement)을 필요로 한 경우, 관련자들에게 양식을 나눠주고 개개인이 그 내용을 기입하는 형식으로 측정데이터를 얻을 수 있다. 그러나 이 때 개개인의 이해관계에 의해 그 결과에 대한 수치가 올바르지 않을 수도 있고, 이러한 문제가 문화자체의 문제로 확산될 수도 있다. 따라서 이러한 두 가지 사례의 예에서 볼 수 있듯이 사람이 할 수 있는 일들의 일부를 자동화할 경우, 프로젝트 실행 시의 오류를 줄일 수 있고, 또한 측정을 위한 기반 데이터들을 쉽게 얻을 수 있다 (manual implementation vs. automation)

이미 제조분야에서는 프로세스 자동화가 생산성을 높이는데 큰 기여를 했다. 따라서 소프트웨어 분야에서도 유사한 방법을 도입하여 생산성을 높이자는 것이 바로 프로세스 자동화 도구에 대한 기본 아이디어이다.

프로세스 자동화는 다음과 같은 문제를 관리하는데 도움을 줄 수 있는 기술이 될 수 있다.

- 프로젝트를 구성하는 활동들의 순서를 가이드 하는 것과 관련된 문제

- 개발되는 산출물들을 관리하는 것과 관련된 문제
- 개발자들이 작업할 때 필요한 도구들을 수행하는 것과 관련된 문제
- 사람의 간섭이 필요하지 않은 자동화 행위를 수행하는 것과 관련된 문제
- 제품을 만드는 사람들 간의 대화를 허용하는 것과 관련된 문제
- 자동적으로 메트릭 데이터를 수집하는 것과 관련된 문제
- 개인의 작업관리를 지원하는 것과 관련된 문제
- 사람의 오류를 줄이는 것과 관련된 문제
- 현재 정확한 상태정보를 가지고 프로젝트 관리를 하는 것과 관련된 문제

프로세스 자동화가 잘 정의된 프로세스를 갖는 프로젝트 작업들을 요구하기 때문에 운영절차의 분명한 이해가 매우 중요하고 선행 되어야 한다. 그리고 보다 효과적으로 수행되기 위해 프로세스 자동화는 프로세스 개선이라는 관점 내에서 이루어져야 한다. 프로세스 자동화 도구는 일반 CASE 도구와는 달리 어떤 조직의 행위에 영향을 준다는 점에서 다르게 분류된다.

1.2 관련된 용어와 개념[2]

1.2.1 관련 용어

1) 프로세스

목적을 성취하기 위해 의도되는, 부분적으로 순서화된 작업들의 집합(A set of partially ordered steps intended to reach a goal). 프로세스라고 하는 용어는 매우 다양한 환경에서 사용되지만, 상기의 정의는 소프트웨어 분야에서만 적용되는 것으로 한정하도록 한다. 소프트웨어 개발을 위한 프로세스의 목적은 보통 소프트웨어 제품 생산, 서비스 제공 또는 증진이 된다. 다른 예로, 소프트웨어 유지보수 프로세스, 인수 테스트 프로세스, 도는 프로세스 개발 프로세스 들을 들 수 있다.

2) 프로세스 자동화

프로세스의 실행 시 기계 프로세스 에이전트(machine process agent) 사용하는 것을 의미한다. 여기서 기계 에이전트의 사용은 적합한 프로세스 프로그램에 포함되어 있는 프로세스 정의에 의해 촉진된다.

3) 프로세스 정의(process definition)

실행(enactable)가능한 프로세스 작업들의 순서화된 형태로 프로세스 디자인을 구현하는 것 (The implementation of a process design in the form of a partially ordered set of process steps that is enactable). 각 프로세스 작업들은 보다 자세한 수준의 작업들

로 세분화될 수 있다. 프로세스 정의는 동시에 실행될 수 있는 프로세스들로 구성될 수 있다. 추상화 수준이 매우 낮아 상세하게 기술 되어진 프로세스 정의는 실행에 적합할 수 있다. 각 프로세스 정의는 해당되는 환경에서 사용되는데, 그 단위는 프로젝트의 종류, 특별한 프로젝트 팀, 그리고 개인 전문가가 될 수 있다.

4) 실행 가능한 프로세스(enactable process)
실행에 필요한 모든 요소들을 포함하고 있는 프로세스 정의의 인스턴스 (An instance of a process definition that includes all the elements required for enactment). 실행가능한 프로세스는 프로세스 정의, 요구되는 프로세스 입력들, 할당된 실행하는 사람들과 자원들, 초기 실행 상태, 초기 실행자와 지속과 종료를 위한 능력 등으로 구성되어 있다. 이러한 능력들이 부족한 프로세스는 실행 가능하지 않다.

5) 프로세스 모델(process model)
프로세스의 아키텍쳐, 디자인, 또는 정의의 추상적인 표현(abstract representation of a process architecture, design, or definition). 프로세스 모델은 분석될 수 있고, 검증될 수 있고, 만약 실행 가능하다면 모델링 된 프로세스를 시뮬레이션 할 수 있다. 프로세스 모델들은 프로세스 분석을 돕기 위해, 프로세스의 이해 및 프로세스 행위를 예측하는데 사용될 수 있다.

6) 프로세스 프로그램(process program)
기계에 의해 실행되기 위해 적합하게 디자인되고 인스턴스화 되어진 프로세스 정의(A process definition which is suitably designed and instantiated for enactment by machine). 프로세스 프로그램은 형식 때문에 특별한 컴퓨팅 환경의 요구에 적합하도록 디자인 되어져야 하고, 컴퓨터 프로그램들처럼 테스팅되고 수정되어져야 한다.

1.2.2 관련 개념

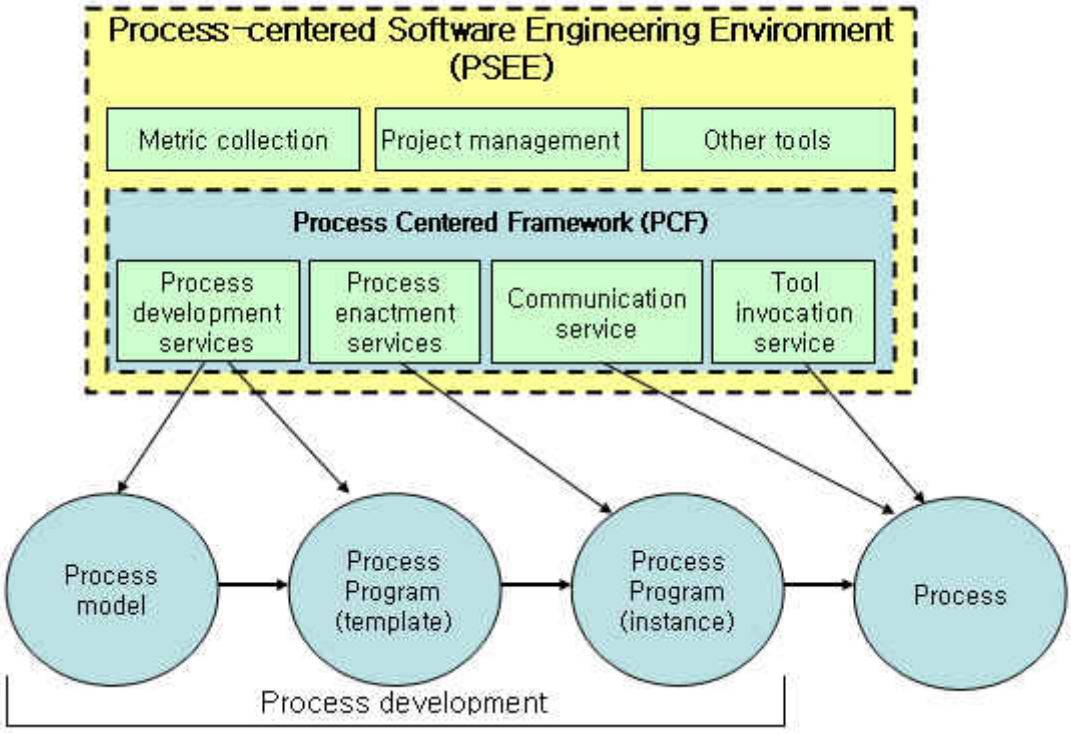
1) 프로세스 중심 프레임워크(Process centered framework : PCF)
프로세스의 정의와 실행에 필요한 기능들을 제공하는 소프트웨어 제품 (a software product which provides the functionality necessary for the definition and enactment of process). 이것은 프로세스 실행언어, 프로세스 실행 메커니즘, 관련된 도구를 수행시키고 사람들과 도구들 간의 의사소통을 제공하는 수단, 그리고 프로세스 실행언어로 기술되어진 프로세스 모델들의 디버깅 기능을 가지고 있다.

2) PSEE
PCF를 수월하게 하는 기능들을 포함하고 여러 어플리케이션 도구들을 포함하는 소프트

웨어 제품(a software product which includes the facilities of a PCF, and also includes application tools)을 의미한다. 지원되는 어플리케이션 도구들로는 메드릭 지원도구, 프로젝트 관리 도구, 소프트웨어 개발 도구 등이 있다.

3) 프로세스 템플릿 (process template)
변수들이 인스턴스화 되지 않은 프로세스 프로그램을 의미한다.

PCF와 PSEE는 두 가지 모두 프로세스 프로그래밍 언어를 지원하는 등의 기본적인 기능들을 가지고 있어야 하지만, PSEE만이 제품의 일부로서 최종 사용자 어플리케이션을 제공하고 있다. 그러나 두 가지 모두 프로세스 내의 작업과 관련된 도구들을 실행(invoke)시키는 기능들은 제공하고 있다.



[그림 1.2.1 : PCF와 PSEE의 개념]

Conradi와 Fuggetta가 “Concepts for Evolving Software Processes[2]”에서 제안한 소프트웨어 프로세스 기술을 위한 참조 프레임워크(reference framework)의 기본은 “Software process는 software production process, software meta-process, software process support라는 세 가지 부분으로 이루어져 있다”는 것이다.

1) Software production process

Software를 만들고, 출시하고, 유지보수 하는데 필요한 모든 활동들을 포함하고 있다.

2) Software meta-process

모든 소프트웨어 프로세스의 진화와 관련된 모든 활동들을 포함한다. 이것의 목표는 출시된 소프트웨어, 그 소프트웨어를 생산한 프로세스, 그리고 메타 프로세스 자체의 품질을 향상시킬 수 있는 모든 혁신들을 소프트웨어 프로세스에 소개하는 것이다.

3) Software process support

소프트웨어 프로세스 활동들의 자동화를 지원해주는 컴퓨터화된 환경을 의미한다.

소프트웨어 프로세스를 모델링 하는 것은 프로세스를 보다 효율적으로 사용하기 위한 것으로 프로세스의 재사용, 자동화, 그리고 관리를 가능하게 하기 위해 필요하다. 앞서 설명한 관련 개념들은 위의 참조 프레임워크의 세 가지 구성요소에 대한 개념들이 함께 존재해야 비로소 의미를 갖게 된다.

2. 프로세스 모델링 언어와 PSEE

2.1 EPOS : Object-oriented Cooperative Process Modeling

2.1.1 개발 배경 및 특징

EPOS는 Expert system for Programming and ("Og") System development를 뜻하는 용어로서 객체 지향 원리(Principle)를 기반으로 하는 프로세스 모델링에 초점을 맞춘 커널(Kernel) 소프트웨어 공학 환경으로 Reidar Conradi(Norwegian Institute of Technology) 등에 의해 개발되었다. EPOS는 크게 EPOS-PM과 EPOSDB의 두 가지 구성 요소로 나뉘어진다. EPOS-PM은 객체 지향 프로세스 모델링 언어인 SPELL과 프로세스 관련 도구(SPELL Interpreter, Execution Manager, Planner, Schedule Manager)로 이루어져 있다. EPOSDB는 형상 관리를 위한 클라이언트-서버 데이터베이스이다. EPOS는 이 EPOSDB라는 데이터베이스를 핵심으로 하여 PSEE 환경에서의 모든 도구들이 이를 기반으로 일련의 층(layer)을 형성하고 있는 것이 특징이다.

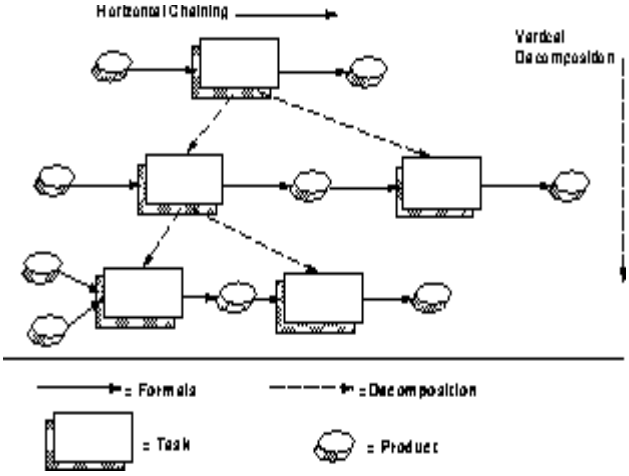
2.1.2 EPOS를 이용한 프로세스 정의와 실행

2.1.2.1 EPOS의 프로세스에 대한 관점

EPOS는 프로세스를 "사용자의 요구사항을 소프트웨어로 변환하는데 필요한 소프트웨어 공학적 활동(Activity)과 관련된 정보들의 집합"이라고 정의하고 있다. 또한 EPOS에서는 소프트웨어 프로세스를 구성하는 요소(Element)들로 활동(Activity), 이와 관련된 산출물(Product), 종속 관계(Dependency), 도구(Application tool), 사람의 역할(Human role), 사용자(User), 조직(organization), 계획(Project)등으로 정의하고 있다.

2.1.2.2 EPOS의 프로세스 정의와 실행

EPOS 에서의 프로세스 모델은 '하나의 외부 프로세스에 관한 내부적인 컴퓨터 기술'이라고 정의하고 있다. 내부 프로세스 모델은 기본적으로 활동에 관한 기술(description)이 그림 2.1.1 과 같이 나뉘어져서(decomposed) 얽혀있는(chained) 하나의 네트워크이다.



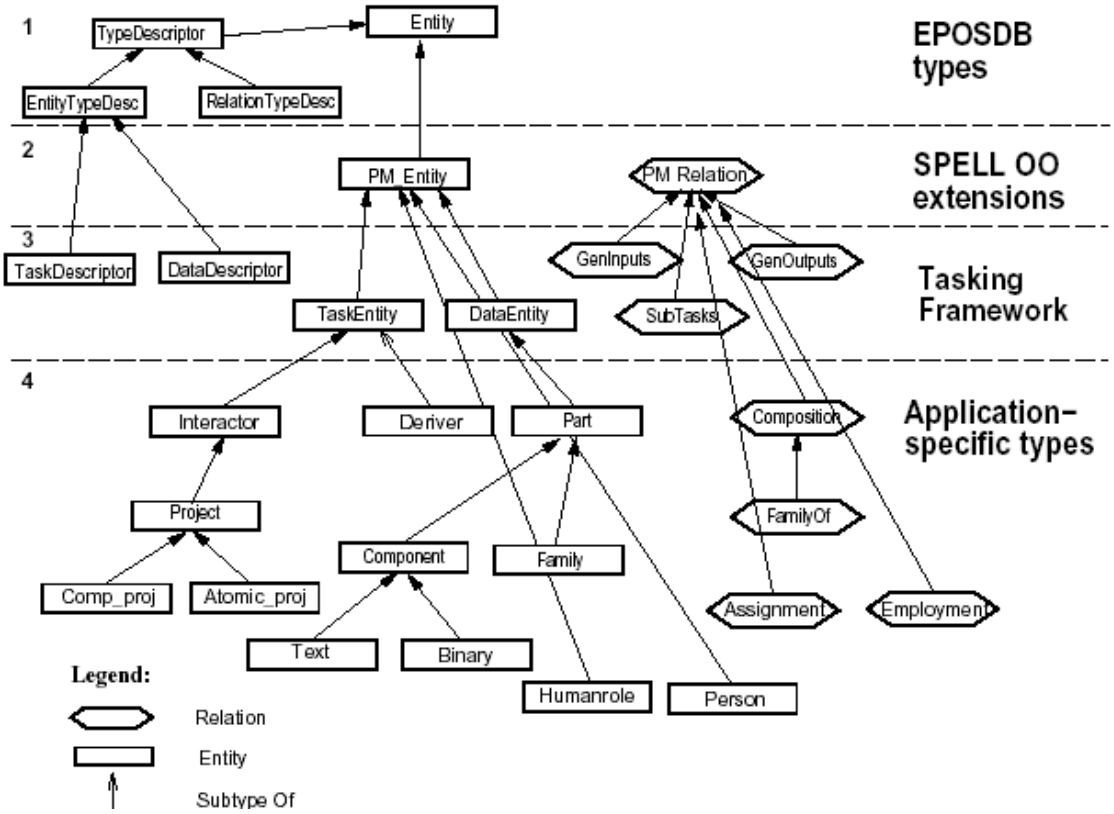
[그림 2.1.1 : 작업(task)의 기본구조]

작업(task)은 일반적으로 모든 형태의 프로젝트 관리 활동을 의미한다. 사람에 의한 활동과 도구에 의한 활성화 모두 작업으로 표현된다. 산출물(product)은 어떤 프로세스에 의해 산출된 소프트웨어 아이템을 말한다. 여기서 우리는 산출물과 활동(activity)이 연계되어 있다고 말할 수 있다. 왜냐하면 모든 활동은 각각 특정 산출물이 필요하고 또한 그 활동으로 인해 다른 활동에서 필요한 어떤 산출물을 생산하기 때문이다. EPOS 의 객체지향 프로세스 모델링 언어인 SPELL 에서의 작업 타입(type)은 소프트웨어 프로세스에서의 기본적인 또는 복합된 단계와, 그 프로세스를 실행하기 위해 필요한 도구, 에이전트(agent), 그리고 그 역할에 관한 지식을 표현한다. 또한 타입의 활성화와 표현된 활동을 수행하기 위해 필요한 코드(또는 스크립트)에 필요한 정보들도 포함하고 있다. 타입은 CODE 부분에 정적인 그리고 동적인 전제조건(pre-condition)과 종료조건(post-condition)을 정의한다. 정적인 전제, 종료 조건은 스크립트를 실행시키지 않고 순방향(forward)과 역방향(backward)으로 추론(reasoning)을 하는데 이용된다. 동적인 전제, 종료 조건은 작업의 동적인 트리거(trigger)를 할 때 쓰인다. 작업은 기본적으로 활성화되어 있고, 작업 네트워크 또는 계획(plan)안에 존재한다.

작업의 실행은 작업 노드의 어느 곳에서도 일어날 수 있다. 작업의 실행은 자동 또는 수동으로 실행될 수 있다. 작업의 순차적인 코드는 그 작업의 동적인 종료 조건이 참(true)이 되게 하고 동시에 다른 작업의 동적인 전제 조건을 만족하여 그 작업이 실행되는 과정을 반복함으로서 일련의 작업이 진행된다.

2.1.3 SPELL

2.1.3.1 타입과 모델 구성



[그림 2.1.2 : EPOS 타입 분류]

그림 2.1.2는 EPOS의 타입들을 보여주고 있다. EPOSDB에는 객체-관계 모델과 유사한 데이터 모델을 가지고 있다. 엔티티(또는 객체)는 유일(unique)하고 변하지 않는 정체성(identity)을 지닌다. 타입 관계의 최상위에는 시스템에서 정의된 Entity라는 타입이 TypeDescriptor라는 메타 타입과 함께 존재한다. 또한 임의의 파일을 저장하는 Longfield라는 타입도 있다.

또한 SPELL은 인스턴스와 타입이라는 두 가지 레벨에서 여러 정의를 지원하고 있다. 즉, SPELL은 타입 레벨에서의 속성(attribute)과 인스턴스/타입 레벨에서의 프로시저(procedure)와 트리거를 제공한다. 메타 타입은 타입 레벨의 정보를 저장할 때 사용된다. 타입 레벨에서의 관계들은 명시적으로 모델링되고 subtyping을 모델링하거나 FORMALS와 DECOMPOSITION과 같은 타입 생성자를 암호화(encode)하는 어떤 내부적인 관계들을 모델링 할 때 사용된다. 또한 하위 타입으로는 PM_Entity와 TaskEntity, DataEntity등이 있다. TaskEntity와 DataEntity의 메타 타입은 각각 TaskDescriptor와 DataDesriptor이다.

2.1.3.2 상속과 보호

동적 바인딩에 의한 단일 상속은 모든 타입 특성에 대해 제공된다. 하나의 하위 타입은

타입 레벨의 속성과 인스턴스/타입 레벨에서의 프로시저를 재정의할 때 사용된다. 타입 레벨의 특성과 redefine, append, concatenate와 같은 프로시저에는 세가지 종류의 상속이 가능하다. 하위 타입에서 재정의된 프로시저는 redefine이나 concatenate에 의해 상속될 수 있다. 속성과 프로시저는 private이나 public으로 선언될 수 있다.

```
ENTITY_TYPE TaskEntity:Entity {
    INSTANCE_LEVEL
        ATTRIBUTES
            Exec-state: …   := … ;            % For task implementation
            Taskstate: String := 'Created';   % Similarly
        PROCEDURES
            i_convert(SelfO:OID, NewT:TID) = …;
            i_delete(SelfO:OID) = …;
            start(SelfO:OID) = …;
            restart(SelfO:OID) = …;
        TRIGGERS
            ON-PROC = … WHEN = AFTER
            COND = … ACTION = …
    TYPE_LEVEL
        ATTRIBUTES
            PRE_STATIC(Inher=append)          Pred = true
            PRE_DYNAMIC(Inher=append)         Pred = true
            CODE          (Inher=concatenate)  Prog = … INNER …
            POST_STATIC(Inher=append)         Pred = true
            POST_DYNAMIC(Inher=append)        Pred = true
            FORMALS       (Inher=redef)       String = 'in:$DataEntity(read) ->
                                                 out:$DataEntity(read/write)'
            DECOMPOSITION(Inher=redef)        String='REPERTOIRE(TaskEntity)'
            EXECUTOR      (Inher=redef)       String='<Logical tool name,
                                                      e.g. CC>'
            ROLE          (Inher=redef)       String='<Logical role name,
                                                      e.g. Designer.>'
        PROCEDURES
            t_create(SelfT:TID, ChildT:TID, <Defined properties>)=…;
                                               % Called by Schema Manager.
            t_delete(SelfT:TID)=…;
            t_change(…)=…;                     % Same parameters as t_create
            i_create(SelfT:TID)=…;             % Normal New generator.
            subgoals(…)=…;
} %TaskEneity-type
```

[그림 2.1.3 : TaskEntity 타입]

2.1.3.3 TaskEntity 타입

기본 작업 타입인 TaskEntity타입의 정의는 그림 2.1.3과 같다. 각 속성들은 String이나 Pred(icate)과 같은 도메인을 가지고 있다. 하나의 작업은 그 타입의 정적, 동적인 행위를 조절하기 위해 많은 수의 타입 레벨 속성을 가지고 있다.

Design이나 Compile과 같은 empty DECOMPOSITION을 가지는 작업의 하위 타입들은 하위 레벨이나 단순한 도구를 암호화한다. Project나 Develop과 같이 non-empty DECOMPOSITION을 가지는 하위 타입들은 보다 높은 레벨의 활동을 암호화한다.

관련된 관계 타입은 작업 인스턴스 분리를 표현하기 위한 SubTasks, 그리고 하나의 작

업 인스턴스들의 실제 입/출력 매개변수와 연결해주는 GenInputs, GenOutputs가 있다.

2.1.3.4 기 정의된 산출물 모델 타입

그림 2.1.2에서의 DataEntity 하위 타입과 그와 관련된 타입들은 EPOS의 산출물 모델 체계를 나타낸다.
Part 타입은 임의의 소프트웨어 부분을 기술할 때 사용된다. Family 하위 타입은 하나의 모듈뿐만 아니라 서브시스템의 묶음을 표현하는데도 사용된다. FamilyOf 관계 타입은 Composition의 하위 타입인데, Family들을 서로 연결시켜준다.
Part 타입의 또 다른 하위 타입인 Component 타입은 소프트웨어 산출물의 atomic한 부분을 표현한다. 이 타입에는 내용, 파일 포맷 등을 표현하기 위한 하위 타입이 존재한다. 이들 타입의 인스턴스는 소프트웨어 부분의 실제 내용을 저장하는 LongField 인스턴스와 관련되어 있다.
Interface와 Body는 ImplementedBy 관계 타입에 의해 연결되어 있는 Component의 간접적인 하위 타입이다. DependsOn 관계 타입은 임의의 두 Component 하위 타입을 연결시켜준다.

2.1.4 도구

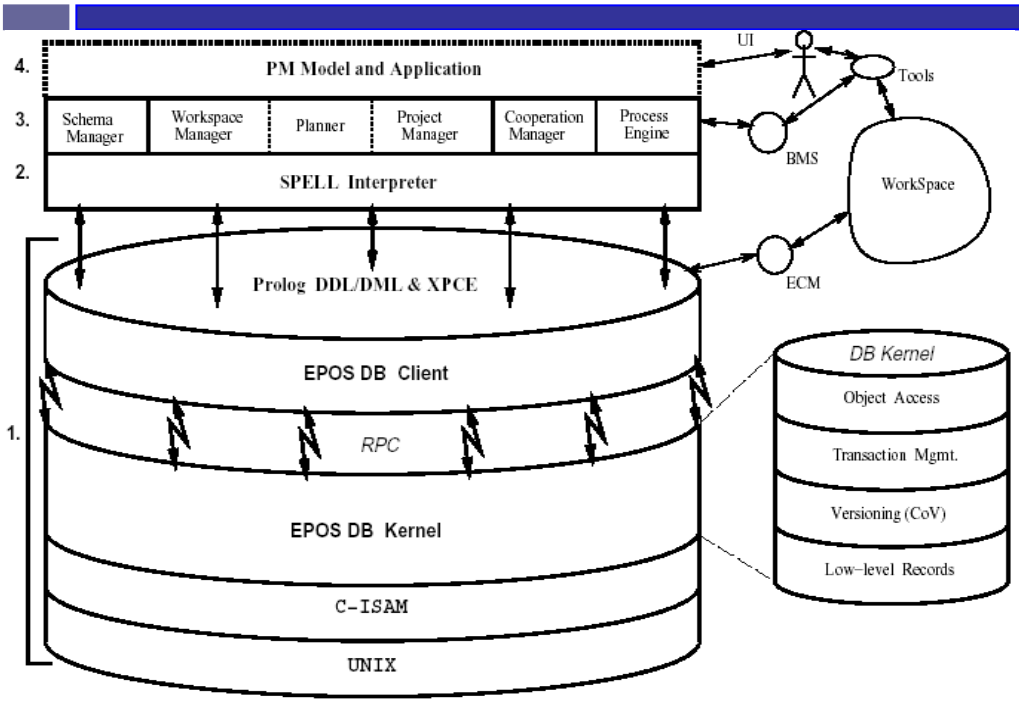
2.1.3.1 아키텍쳐

그림 2.1.4는 EPOSDB와 이를 기반으로 층을 형성하는 도구들의 아키텍쳐를 보여주고 있다. EPOS의 층은 크게 다음과 같이 나눌 수 있다.

1) EPOSDB : 균일(uniform)하고 변화 지향적인 버전관리(change-oriented versioning) 및 구조적인 객체 지향 데이터 모델을 제공하는 클라이언트-서버 구조의 DBMS이다.

2) 객체 지향 프로세스 모델링 언어 : SPELL

3) 작업 프레임웍 : 관련된 도구와 사람에 의한 작업 네트워크의 정의와 동시 실행을 위한 프레임웍이다.



[그림 2.1.4 : EPOSDB와 클라이언트 도구]

4) 특정 도메인 또는 응용 프로그램에 한정된 프로세스 모델 : 작업 타입과 같은 프로세스 체계(Process Schema)를 가지고 있고, 도구의 활성화등과 같은 인스턴스를 가지고 있다.

2.1.3.2 EPOS 프로세스 도구

1) SPELL Interpreter : SPELL로 기술된 프로세스 모델을 해석한다

2) Execution Manager : SPELL Interpreter와 함께 EPOS 프로세스 엔진을 이루는 도구로, SPELL Interpreter에서 해석된 작업 타입을 실행한다.

3) Planner : 특정 작업에 대해서 그 작업의 하위 작업을 생성한다.

4) Schema Manager : 정의된 타입들을 수정하고 탐색하는 등의 관리를 한다.

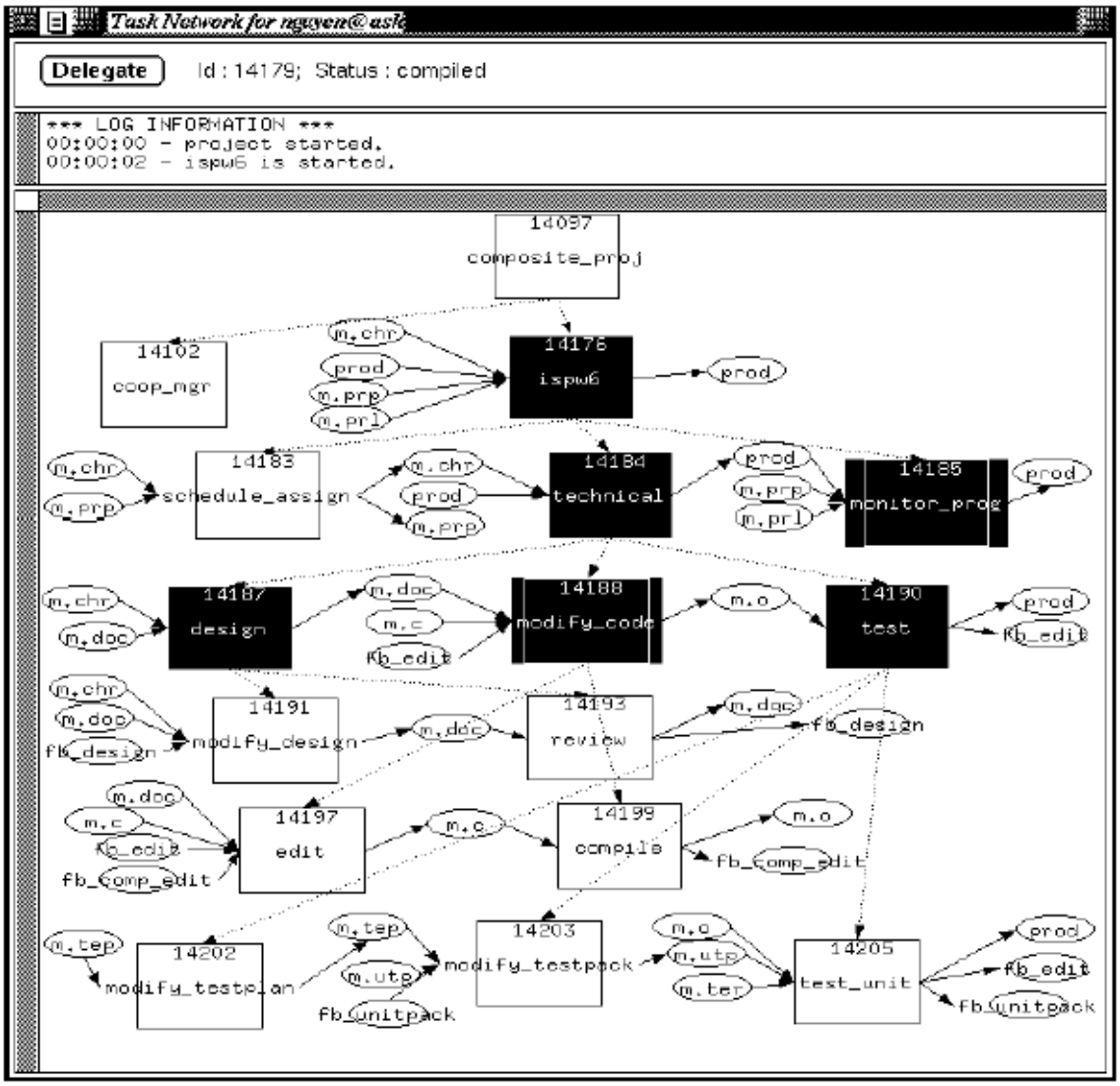
5) Workspace Manager : 데이터베이스에 저장된 파일과 실제 모델링을 하는 workspace간의 check-in, check-out에 관련된 작업을 수행한다.

6) Cooperation Manager : 상호 협동하는 트랜잭션(Cooperating transaction)간의 하위 레벨 통신 프로토콜을 설정하고 유지한다.

7) Project Manager : 여러 하위 작업들간의 의존성을 줄여 줌으로서 여러 개발자들간의 수작업에 의한 상호 협동을 줄여준다.

2.1.5 예제

EPOS-PM을 이용하여 모델링하는 스냅샷은 그림 2.1.5와 같다.



[그림 2.1.5 : EPOS-PM의 화면 예]

2.2 SOCCA : Specifications of Coordinated and Cooperative Activities

2.2.1 개발배경 및 특징

소프트웨어 프로세스를 모델링하는 다중-패러다임(Multi-paradigm) 접근법들이 많이 제안되어 왔다. 이들은 모두 하나의 주요 패러다임에 기대고 있는데, 주로 규칙(Rules), 명령형 프로그램(Imperative program) 또는 패트리넷(Petri-Net)이다. 이런 접근법들은 두 가지 특징이 있는데 첫째는 주로 텍스트(Text)를 기반으로 하는 기술 기법을 가졌다는 것이고, 둘째는 대부분 프로그래밍 언어 수준의 추상화 수준을 제공한다는 것이다. 이 특징은 디자이너가 문제 자체를 직접 모델링하기보다 모델링 형식(Formalism)을 위한 기술적인 내용에 더 치중하도록 만든다. 또, 이 접근법들은 소프트웨어 프로세스의 다양한 관점들(Perspectives)을 분리해서 모델링하는 방법을 제공하고 있지 않다.

소프트웨어 프로세스 모델링에 대한 복잡도(Complexity)와 요구사항들을 고려했을 때, 모델링 기법에서 세 가지 중요한 고려 사항이 있다. 첫째는 하위 수준(Low-level)의 모델링 또는 프로세스의 구현에 앞서서 상위 수준(High-level) 명세를 기술하는 단계가 필요하다는 것이다. 둘째는 소프트웨어 프로세스가 필요한 모든 관점에서 분리되어 모델링 될 수 있어야 하고 또 이것을 완전한 명세로 통합하는 방법이 필요하다는 것이다. 셋째는 각 모델에서 인적 부분(Human part)이 명시적으로 나타날 수 있어야 한다는 것이다.

SOCCA(Specification of Coordinated and Cooperative Activities)는 인적 부분(Human part)과 나머지 부분들 사이의 상호작용을 명시적으로 포함하는 소프트웨어 프로세스 모델링을 위한 명세 형식(Formalism)이다. 또한, SOCCA는 세가지 관점으로 프로세스를 모델링하는데 이것은 데이터 관점, 행위(Behavior) 관점, 프로세스 관점이다. 각 관점을 모델링하기 위해 확장된 엔터티-관계 모델링(Extended entity-relationship modeling)을 기반으로 하는 객체 지향 모델링 기법, 상태 전이 다이어그램(State transition diagram)과 PARADIGM, 객체 흐름 다이어그램(Object flow diagram)을 이용한다.

2.2.2 SOCCA를 이용한 프로세스 정의와 실행

2.2.2.1 SOCCA의 프로세스에 대한 관점

SOCCA의 프로세스에 대한 관점은 다음의 정의된 엔터티들로 요약될 수 있다.

- 프로젝트 팀(Project team)
  프로젝트를 수행하는 팀 내 인적 자원을 말한다.
  (예: project manager, design engineers, quality assurance engineers 등)
- 프로젝트 문서(Project documents)
  프로젝트 진행 중에 생성되는 문서를 말한다.

(예: design part, code part, test plan, test package 등)

- 작업(Task)

프로젝트 진행 중에 프로젝트 팀원들이 해야 할 작업을 말한다.

(예: schedule and assign tasks, modify design, monitor progress 등)

그리고, 위의 엔터티들을 기준으로 하여 데이터, 행위, 프로세스 관점이라는 또 다른 시각으로 바라보고 모델링 하게 된다.

2.2.2.2 SOCCA의 프로세스 정의와 실행

소프트웨어 프로세스 모델의 복잡도와 크기 때문에 결과적으로 이것을 명세한 결과 또한 복잡하고 크다. 이 때 여러 개의 관점별로 따로 모델링하는 방법은 모듈화(Modularity)를 향상시켜 결국 모델을 변경하는 등 유지보수를 쉽게 하는 효과를 준다. 이런 시각에서 SOCCA는 세 가지 관점에서 프로세스를 모델링한다. 세가지 관점은 데이터 관점, 프로세스 관점 그리고 행위 관점이다.

각각의 관점을 모델링하기 위해 세 가지 명세 형식(Formalism)을 이용한다. 먼저 데이터 관점은 시스템의 정적 구조에 초점을 맞춘다. 이 때 사용되는 명세 형식은 확장된 엔터티-관계 모델링(Extended entity-relationship modeling)을 기반으로 하는 객체 지향 모델링 기법이다. 둘째 프로세스 관점은 시스템의 기능(Functionality)에 초점을 맞추고 이 때는 객체 흐름 다이어그램(Object flow diagram)을 사용한다. 셋째 행위 관점은 알고리즘, 특히 데이터의 변환을 설정하는 스텝 순서에 초점을 맞춘다. 이 때는 상태 전이 다이어그램(State transition diagram)과 PARADIGM을 사용한다. PARADIGM은 원래 병렬 프로세스(Parallel process)를 명세하기 위해 개발된 명세 형식이다.

이 세가지 관점의 모델을 통합하는 방법의 기본은 하나의 명세 형식으로 기술된 내용에서 다음 명세 형식으로 기술된 내용을 기술하기 위해서 전자에서 필요한 구성요소를 주의 깊게 선택하는 것이다. 이것을 위해 관점 뿐 아니라 스케일(Scale)에 따라서도 모델을 나누어 볼 필요가 있다. 이 두 가지가 모두 고려된 결과가 다음 테이블에 기술되어 있다.

	Data	Behavior	Process
Local	class attributes operations	external behavior states transitions	process components inputs/outputs
Nearby	relationships part-of is-a	internal behavior states transitions	(de)composition
Global	relationships	coordinated	flow

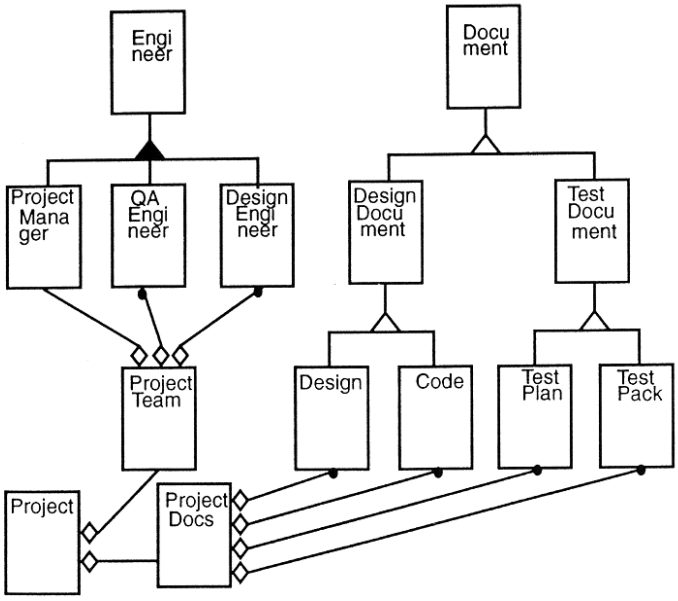
	general uses	behavior manager processes subprocesses traps	objects from/to
SOCCA concepts	Extended entity-relationship model	State transition diagram + PARADIGM	Object flow diagram

[표 2.2.1 : Specification perspectives of different scale]

다음 절부터는 위의 세가지 관점을 사용되는 세가지 모델링 형식과 이들을 통합하는 방법에 대해 설명하도록 한다.

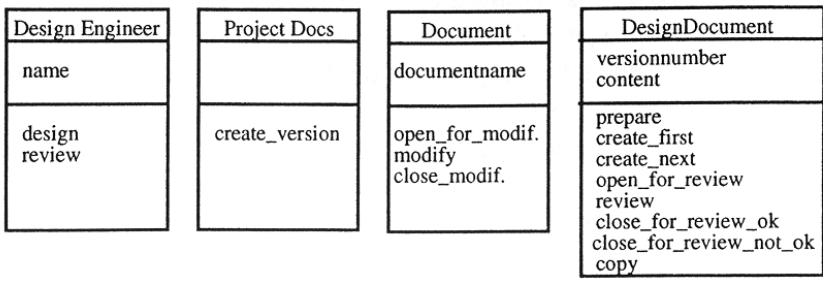
1) 클래스 다이어그램(Class diagrams)

여기서는 클래스 다이어그램의 세 단계로 살펴 보겠다. 첫째 단계는 복잡한 구조의 객체들은 클래스 계층 구조로 표현한다. 객체들 사이의 관계는 part-of 과 is-a 관계인데 이것은 데이터 관점의 local과 nearby 스케일을 커버한다. 둘째 단계는 결여된 일반화 관계를 추가한다. 셋째 단계는 확장된 엔터티-관계 모델링의 부가 요소로서 다른 클래스가 제공하는 오퍼레이션을 어떤 클래스가 사용하는지에 대해 기술한다. 이것은 uses 관계라고 부른다. 이것은 import/export 다이어그램에서 따로 언급된다.
여기서부터는 ISPW-6을 위의 세 단계로 기술한 예를 살펴보겠다. 첫 단계는 ISPW-6을 일부를 클래스 다이어그램으로 표현한 것이다.



[그림 2.2.1 : Class diagram: classes and is-a and part-of relationships]

클래스들은 직사각형으로 표현되어 있다. 가독성을 위해서 클래스의 attribute와 operation은 그림2.1.2에 표현되어 있다.

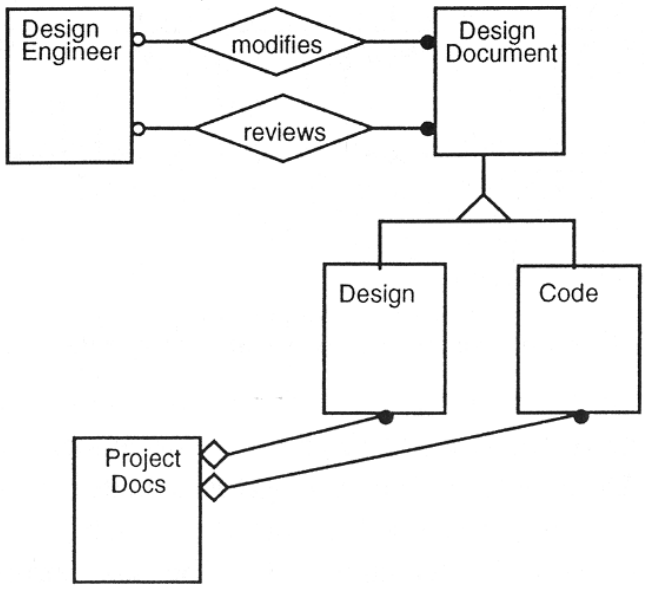


[그림 2.2.2 : Class diagram: attributes and operations]

위 그림 두 개를 동시에 봐야 한다. 더구나, 위의 attribute와 operation은 나중에 전체 SOCCA 명세의 다른 부분에 값을 돌려줄(returned) 것만 명기되어 있다. 그림 2.1.1에서는 작은 삼각형들이 is-a 관계를 작은 다이아몬드들은 part-of 관계를 표시한다. 검은색 삼각형은 서로 따로따로 떨어진 is-a 관계를 표시한다. 예를 들면, Engineer는 ProjectManager와 DesignEngineer로 행위할 수 있다. 검은색 점은 특정 관계에 참여하는 인스턴스의 수가 0개에서 n개 까지 가능하다는 것을 말한다.

둘째 단계는 다른 클래스들을 연결하는 일반화 관계를 추가한다. 위 그림의 경우 다이어그램이 단지 우리 예에 맞는 클래스만 포함하도록 제한하였다. 이 제한된 다이어그램 내에 DesignEngeer, ProjectDocs 그리고 Design이라는 클래스와 modifiers와 reviews

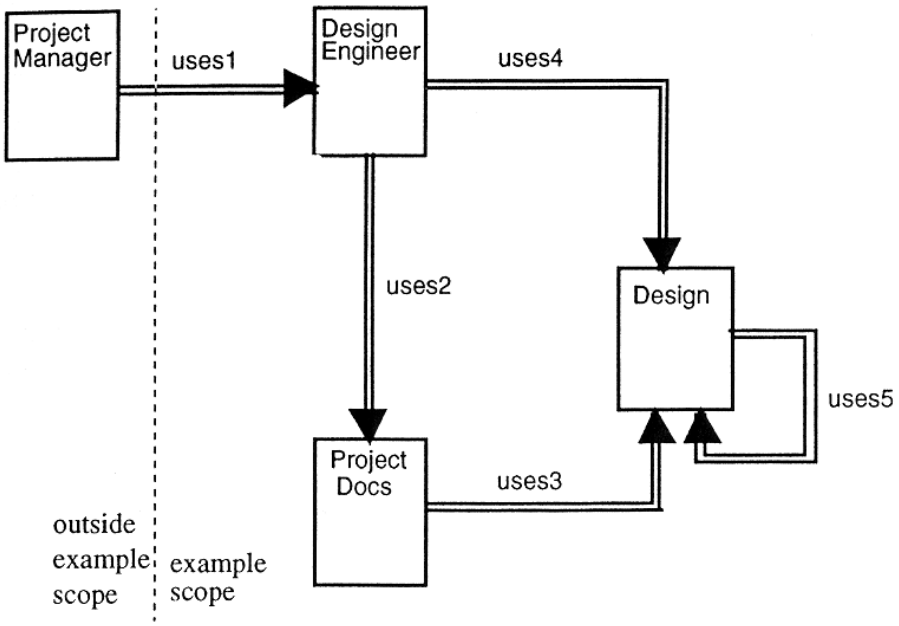
라는 일반화 관계가 포함되어 있다. 이것은 다음 그림에 나타나 있다.



[그림 2.2.3 : Class diagram: classes and general relationships]

속이 빈 점은 이 관계에 참여하는 객체가 0개 또는 한 개라는 것을 표현한다.

셋째 단계는 더 객체 명세를 확장한다. 이것은 첫째 단계에서 생성된 클래스 다이어그램에 확장된 엔터티-관계 모델링에 일반적으로 주어져 있지 않은 정보를 추가함으로써 이루어진다. 새로운 이진 관계 형식인 uses는 다양한 export operation이 어디서 import 되는지를 명시한다. 이 단계의 결과는 import/export 다이어그램으로 표현되는데 다음 그림과 같다.



[그림 2.2.4 : Import/export diagram]

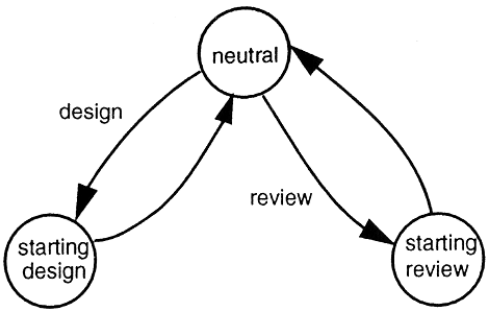
각 uses 관계는 attribute를 가지는데, 이것은 import_list라고 부른다. 주로 import되는 operation의 이름을 리스트로 보관할 때 사용된다. 이 리스트는 같은 형식이라도 서로 다른 인스턴트들 사이에서는 다른데 이것은 이 인스턴스의 역할에 달려있다. 예를 들면, DesignEngineer는 Design으로부터 다른 operation을 필요로 하기도 하고, export하기도 하는데 이것은 역할이 designer인지, reviewer인지에 따라 달라진다.

2) 상태 전이 다이어그램(State transition diagrams)과 PARADIGM

앞에서 다룬 클래스 다이어그램에서는 operation의 역학(dynamics)이나 행위(behavior)에 대해서는 기술하지 않았다. 상태 전이 다이어그램은 특정 클래스의 로컬 행위(local behavior)에 초점을 맞춘다. 로컬 행위는 두 가지 다른 행위로 구분되는데, 그것은 외부 행위(external behavior)와 내부 행위(internal behavior)이다. 외부 행위는 외부에서 보이는 것으로 export된 operation들을 부르는 허용된 순서들을 명시한다. 내부 행위는 private operation들과 다른 곳에서 import된 operation을 부르는 허용된 순서들을 명시한다. 이 행위들을 기술할 때는 세가지 단계를 통한다. 첫째 외부 행위는 각 클래스별로 명세된다. 둘째 각 클래스에 대해서 그것이 다른 곳에서 어떤 operation을 import하는지 명시한다. 셋째 각 operation의 내부 행위를 명세한다.
이제부터 예를 통해서 상태 전이 다이어그램을 통해서 행위를 명세하는 방법을 설명하겠다. 여기서는 데이터와 관련된 자세한 부분은 제외시키고 대신 operation들의 순서에 초점을 맞춘다. 먼저 DesignEngineer의 외부 행위를 명세하는 것으로 시작하고 다음으로 ProjectDocs와 Design의 외부 행위를 명세하겠다. DesignEngineer의 export operation은 다음과 같다.

design(doc-name)
review(doc-name)

어떤 디자인 엔지니어도 동시에 여러 디자인이나 여러 리뷰(review)에 참여할 수 있으므로 위의 operation을 부르는데 있어서 확정된 순서는 없다. 이것을 그림으로 나타내면 다음과 같다.

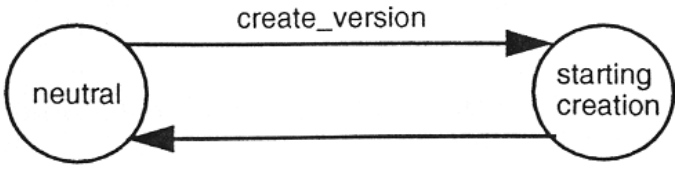


[그림 2.2.5 : Design Engineer: STD of the external behavior]

위 그림에서 neutral 상태는 디자인 엔지니어가 디자인이나 리뷰를 마치면 돌아오는 상태이다. Start design 상태는 디자인 작업을 시작하는 상태로 볼 수 있다. 그런데 neutral 상태로 돌아가는 전이에 라벨이 붙어 있지 않다. 이것은 이 전이에 해당하는 export operation이 없다는 것을 뜻하는데 좀 더 정확히 말하면 이 전이는 단지 (내부) 디자인 작업이 정말 시작된 후에야 일어난다. 이것을 설명하려면 외부행위와 내부 행위 사이의 관련성에 대해 알아야 하는데 이 설명은 PARADIGM을 설명할 때 할 예정이다. 다음으로 ProjectDocs의 export operation을 살펴보면 다음과 같다.

create-version(doc-name)

이것을 상태 전이 다이어그램으로 나타내면 다음과 같다.



[그림 2.2.6 : ProjectDocs: STD of the external behavior]

다음으로 궁극적으로 관심 있는 Design에 대한 외부 행위를 살펴보자. Design의 export operation들은 다음과 같다.

prepare

create-first

create-next

open_for_modif.

modify

close_modif.

open_for_review

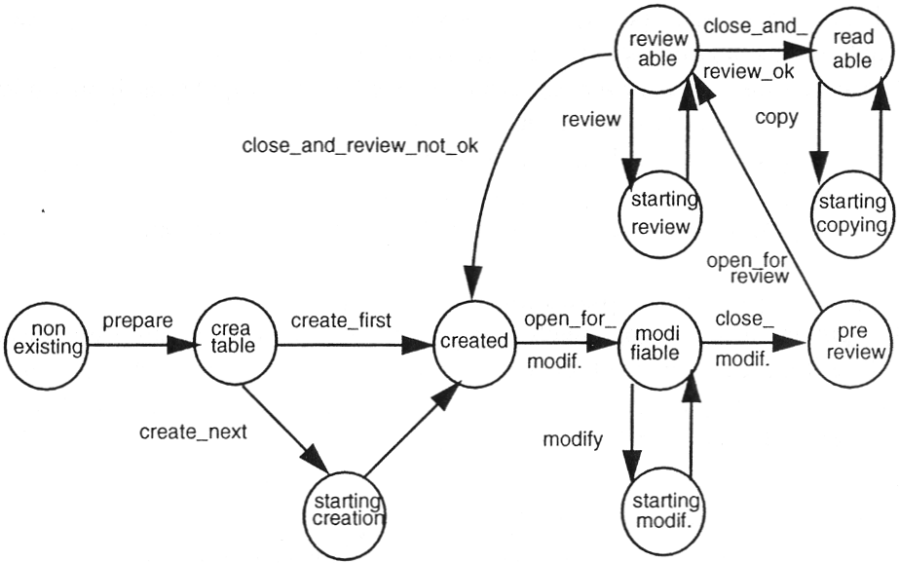
review

close_and_review_OK

close_and_review_not_OK

copy

위의 operation들의 가능한 순서라는 관점에서 Design의 각 인스턴스는 정확히 디자인 문서의 하나의 버전을 표현한다는 것을 명심하자. 이것을 상태 전이 다이어그램으로 나타내면 다음과 같다.

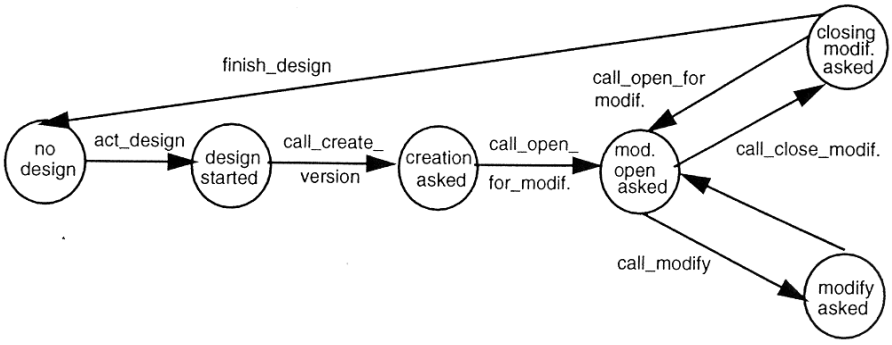


[그림 2.2.6 : Design: STD of the external behavior]

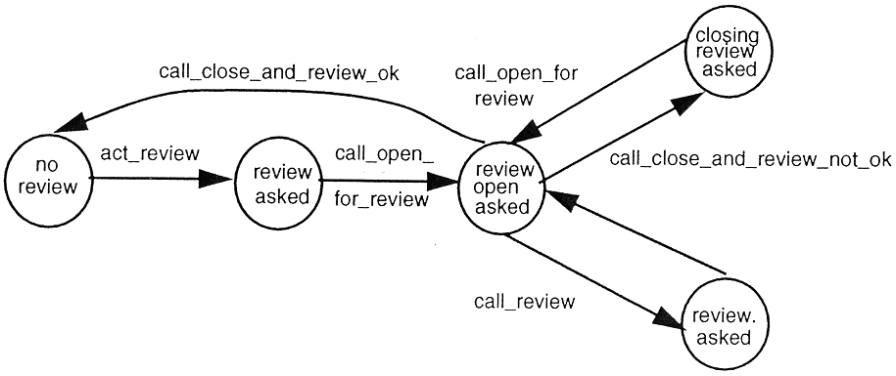
이제부터 각 operation의 내부 행위를 명세해 보자. 먼저 <그림 2.4>에서 각 uses 관계에 대해서 가능한 import된 operation들의 리스트를 나타내어야 한다. 이 리스트는 실제로 uses 관계의 import_list라는 attribute의 값 영역(value domain)을 더 정확히 명시할 수 있게 한다. 실제 리스트는 다음과 같다.

```
uses1
    design (doc-name)
    review (doc-name)
uses2
    create_version (doc-name)
uses3
    create_first
    create_next
uses4
    open_for_modification                % used as designer
    modify
    close_modification
    open_for_review              % used as reviewer
    review
    close_and_review_OK
    close_and_review_not_OK
uses5
    copy
    prepare
```

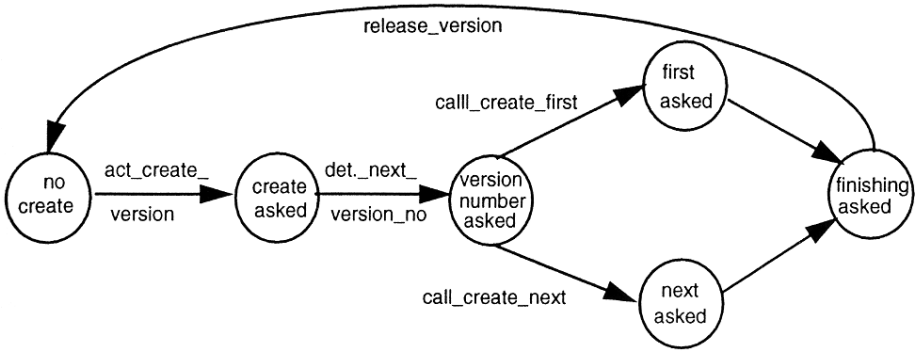
위의 리스트는 가능한 모든 import된 operation들을 가리킨다. 특정 객체의 행위에 따라 단지 위의 operation중에서 일부만 이 실제로 import된다. 예를 들면, 디자이너로서 DesignEngineer는 uses4 관계의 첫 3개의 operation을 필요로 하지만 리뷰 담당자(reviewer)로서는 uses4 관계의 마지막 4개의 operation을 필요로 한다. 이제 실제로 내부 행위에 대해 명세를 해보자. 여기서는 5개의 내부 행위를 명세한다. 이것은 int-design, int-review, int-create_version, int-create_next 그리고 int-OK이다. 어떤 operation의 내부 행위 명세 내에는 두 가지 다른 형식의 operation이 발생할 수 있다. 첫째 형식은 call이라는 prefix로 시작하는 operation으로써 해당하는 export된 operation이 있다. 둘째 형식은 call prefix 없이 사용되는 것으로 단지 내부 operation을 가리킨다. 이들 중 act라는 prefix가 붙은 것이 있는데 이것은 내부 행위와 외부 행위 사이의 내부 통신에 관련된 것이다. 아무 prefix가 없는 것은 내부 행위 내에서 내부적인 것을 말한다. 5개의 내부 행위 명세는 다음 그림들과 같다.



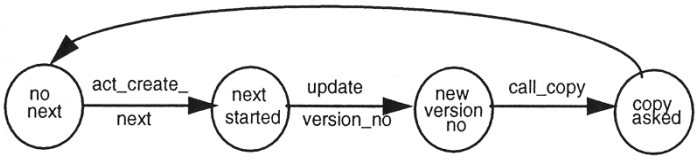
[그림 2.2.7 : int-design: STD of the internal behavior]



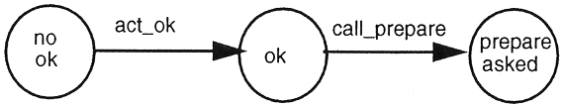
[그림 2.2.8 : int-review: STD of the internal behavior]



[그림 2.2.9 : int-create_version: STD of the internal behavior]



[그림 2.2.10 : int-create_next: STD of the internal behavior]



[그림 2.2.11 : int-OK: STD of the internal behavior]

위 그림에서 보면, Design Engineer의 내부 행위를 나타내는 int-design과 int-review에 DesignEngineer의 모든 export된 operation을 포함하고 있음을 알 수 있다. 비슷한 성질이 ProjectDocs의 오직 하나 뿐인 내부 행위인 int-create_version에서도 성립하고 있음을 알 수 있다. 결국은 Design의 두 개의 내부 행위인 int-create_next와 int-OK은 copy와 prepare라는 operation 두 개를 포함하는데 이것은 Design에서 export되고

Design에 import된 것이다. 또 다른 공통적인 성질은 act를 prefix로 하는 operation으로 시작된다는 것인데 이 operation은 내부 행위의 예비적인 활동을 한다.

지금까지 외부 행위와 내부 행위 각각에 대해서 설명하였다. 이 행위 명세들은 모두 순차적이라는 특징을 가지고 있다. 여기서부터는 이들 사이의 조정(coordination)에 대해서 설명하겠다. 앞에서 언급했던 것과 같이 어떤 export operation의 실행은 대응되는 내부 행위의 실행을 시작하게 한다. 역도 성립한다. 이것을 두 개의 순차적 행위 사이의 통신이라고 하고 두 가지 종류로 나눈다. 첫째는 임의의 내부 행위와 다른 객체의 외부 행위사이에 발생하는 통신을 말한다. 둘째는 같은 객체의 외부 행위와 내부 행위 사이에 발생하는 통신을 말한다. 이 두 종류의 통신을 살펴보면 외부 행위가 다른 객체든 객체 자신에서든 내부 행위들 사이의 중간자 역할을 한다는 점에서 이 둘을 합하여 type-1이라고 분류한다. 그리고 또 다른 분류를 고려하는데, 이 분류는 같은 객체의 모든 내부 또는 외부 행위의 순서, 우선순위, 유사성 의존도를 결정하고 조절하기 위해 발생하는 통신으로 구성된다.

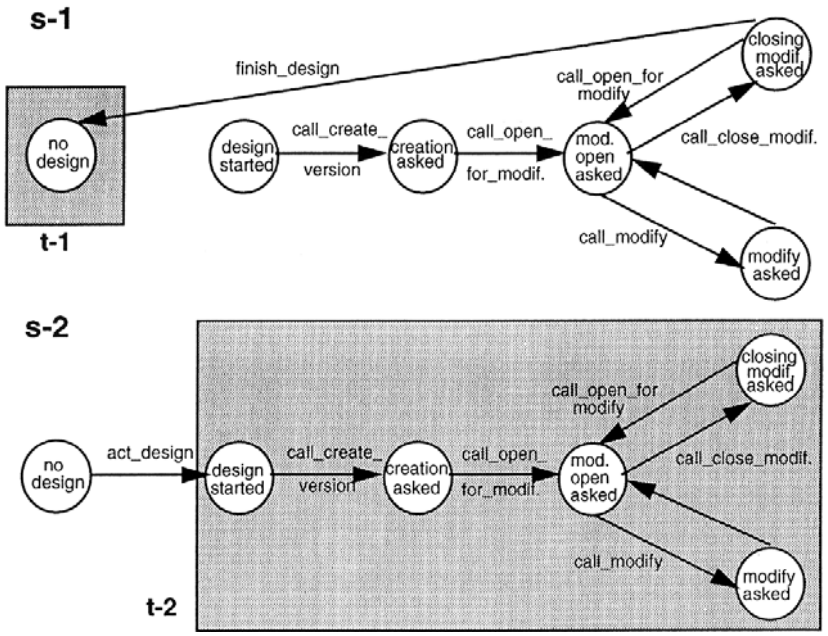
여기서는 subprocess, trap 그리고 manager process의 PARADIGM 개념을 이용한다. PARADIGM은 원래 coordinated parallel processes의 명세를 위해 개발된 것이다. 모델링에 대한 기본적인 아이디어는 다음과 같은 4단계로 기술된다.

(1) 각 프로세세의 순차적인 행위는 상태 전이 다이어그램을 통해서 기술된다.
(2) 각 상태 전이 다이어그램내에서는 중요한 subdiagrams(subprocesses라고 부른다)가 다른 프로세스와의 조정을 고려하여 정해진다.
(3) 각 subprocess내에서 상태들(traps라고 부른다)들이 정해진다. 이것은 객체가 한 subprocess에서 다음으로 교환될 준비가 된 상황을 기술한다.
(4) 모든 객체들, 행위들 사이의 subprocess들 사이의 가능한 전이들은 상태 전이 다이어그램에 의해 더 자세히 기술되는데 이것을 manager process라고 부른다.

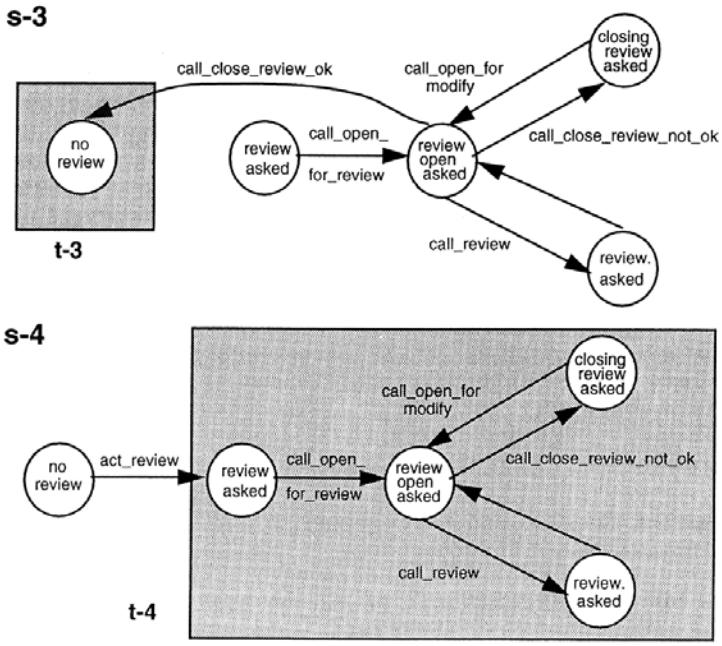
위에서 외부 행위가 다른 객체든 객체 자신에서든 내부 행위들 사이의 중간자 역할을 하는 경우를 언급하였다. 이것은 각 외부 행위 명세를 PARADIGM 모델에서 manager process로 서비스하는 좋은 후보가 될 수 있다는 것을 뜻한다. 이 manager의 고용인(employee)은 한편으로는 import된 operation을 부르는 내부 행위이고 다른 한편으로는 불러진 export operation의 내부 operation을 실행하는 객체 자신의 내부 행위이다. PARADIGM의 형식에 따르면 외부 행위의 각 상태는 행위 제약(behavior restriction) 즉, 각 고용인의 subprocess에 해당된다. 이 대응은 manager가 특정 상태에 있을 때 고용인이 그들의 행위를 이 subprocess에 한정하도록 한다. 이 행위 제약은 현재 행위 제약이라고 불리는데 이것은 manager의 현재 상태에 의존한다. 또, 외부 행위의 각 전이는 특정한 최종 부분 즉, 현재 행위 제약의 trap에 대응된다. 이 대응은 manager가 관련된 고용인들이 이 trap에 들어왔을 때만 전이를 할 수 있음을 뜻한다. 완전한

PARADIGM 명세는 매우 복잡하므로 여기서는 다음 규칙에 따라 예제가 구성된다. 첫째는 manager와 고용인을 중심으로 개괄적인 내용이 기술된다. 둘째, 각 고용인별로 subprocess와 traps이 manager에 대해서 기술된다. 셋째, manager와 고용인의 상태와 전이 사이의 대응이 명시된다. Subprocess와 trap의 그림으로 다음과 같이 나타낸다. 먼저 subprocess는 원래 행위를 제약한 상태 전이 다이어그램으로 나타내고, trap은 이것이 일부분인 상태를 포함하는 어두운 색의 다각형으로 나타낸다.

여기서는 하나의 예제를 통해 PARADIGM을 적용하는 방법을 알아 보겠다. 예제는 DesignEngineer의 외부행위를 PARADIGM을 이용해서 기술한 것이다. 여기서 고용인은 두 개의 내부 행위인 DesignEngineer:int-design과 int-review이다. Int-design과 int-review의 subprocess와 trap은 다음 그림과 같다.

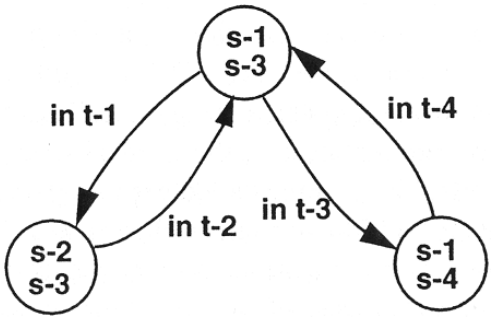


[그림 2.2.12 : int-design's subprocess and traps with respect to DesignEngineer]



[그림 2.1.13 : int-review's subprocess and traps with respect to DesignEngineer]

그리고, manager의 상태와 subprocess 사이의, 또 전이와 traps 사이의 대응은 다음 그림에 기술되어 있다. Subprocess는 상태의 라벨로 나타나 있고, trap은 전이의 라벨로 나타나 있다.



[그림 2.2.14 : int-review's subprocess and traps with respect to DesignEngineer]

주목해서 볼 부분은 trap t-2과 t-4는 manager가 내부 행위 중 하나를 시작하고 나서 바로 neutral 상태로 가능한 빨리 돌아올 수 있도록 선택되었다. 이것은 manager와 내부 행위 사이의 모든 통신에서의 특징이다. 이제 위 그림이 기술하는 행위에 대해서 알아보자. 다른 곳에서 design에 대한 호출이 Name1이라는 이름의 문서를 매개변수로 이루어졌다고 가정하자. 호출의 결과로 DesignEngineer는 neutral에서 starting design으로 전이하기 위해서 초대되었다. 그러나 그 순간 DesignEngineer는 여전히 starting design의 외부에 있다. 이것은 int-design이 그것의 subprocess s-1 내에 있고 또한

trap t-1에 있음을 의미한다. 단지 int-design이 trap t-1에 도달한 후에, DesignEngineer가 neutral에서 starting design으로 전이할 수 있다. Start design 상태에 들어갔을 때 DesignEngineer는 subprocess s-2가 int-design에 도달하기를 명령한다. Int-design이 trap t-1에 있었기 때문에 trap t-2에 들어감과 거의 동시에 no design 상태에서 즉시 s-2 내부로 진행한다. Trap t-2에 들어가자 마자 DesignEngineer는 starting design을 떠나 neutral로 전이한다. Int-design에 대해서 살펴보면, 이것은 s-1 내부에서 차분히 필요한 모든 호출을 다하면서 진행하면서 끝내는 trap t-1으로 들어간다는 것을 뜻한다. 한편으로 t-1에 들어가지 않는 한 DesignEngineer는 어떤 리뷰나 디자인도 시작할 수 있다. 같은 문서 Name1에 대해 작업하는 여러 DesignEngineer들 사이의 조정은 Design으로 표기되고 Name1이라는 매개변수를 가지는 하나의 문서에 대한 외부 행위로 조정된다.

지금까지 type-1 통신에 대해서 논의하였다. 여기서는 type-2 통신의 기술은 하지 않았는데 이유는 이것은 프로세스 모델을 명세보다 프로세스 실행과 관련 있기 때문이다. 게다가 type-2 통신의 발생과 구조는 모델을 기계의 구조 뿐 아니라 인적 조직의 구조와의 매핑에 의존적이다. 이것이 type-2 통신이 프로세스 모델의 실행과 관련 있는 이유이다. 그러나, PARADIGM은 type-2 통신을 기술하는데도 매우 유용하다.

3) 모델들의 통합 및 일관성

이 절에서는 클래스 다이어그램과 PARADIGM 명세사이의 연결에 대해서 설명하겠다. 클래스 다이어그램은 데이터 관점을, PARADIGM 모델은 행위 관점을 표현한다. 클래스 다이어그램에서는 통신이나 객체들 사이의 연결이 관계를 통해서 모델된다. 이것은 관련된 객체들의 데이터 명세 사이에 어떤 연결이 있음을 나타낸다. PARADIGM에서는 다양한 행위들 사이의 통신이 subprocess와 trap으로 모델링된다. 이것은 통신이 행위에 어떤 영향을 주는지 정확히 표현한다.

데이터 기술에서 행위의 영향과 행위 사이의 통신은 두 가지 현상으로 반영된다. 첫째는 객체와 관계 인스턴스의 attribute 값의 변화이고 둘째는 객체나 관계 인스턴스의 추가 또는 삭제이다. 행위 기술에서는 데이터 측면은 상태를 통해서 시각적으로 볼 수 있다. 어떤 행위는 여러 개의 스텝, 전이, 그 전이가 이끄는 상태의 영향으로 구성된다. 이것은 두 관점의 통합이 객체와 관계 인스턴스의 추가, 변경, 삭제라는 입장에서 전이의 양향을 기술함으로써 이루어질 수 있다는 뜻이다.

실제로는 모든 전이가 데이터에 직접적인 영향을 주지는 않는다. 이유는 많은 행위가 다른 행위에 영향을 주려는 목적을 가진 것이 많기 때문이다. 예를 들면 어떤 operation 호출은 비동기적으로 특별한 외부 행위가 이 operation을 시작하도록 요청한다. 다시 이 외부 행위가 비동기적으로 내부 행위 중 하나가 이 영향을 이루도록 요청한다. 다양한 영향이 체계적으로 논의 될 수 있다. 어떤 전이는 “implicitly modeled”(약

자로 i.m.이라 부른다)라는 말을 붙이는데, 이것은 그 전이가 PARADIGM 모델에 명시적으로 표현되지 않는다는 것을 의미한다. 이 전이가 유발하는 내부 행위가 너무 간단해서 나타내지 않는 것이다.

그림 2.2.15는 참가한 상태 전이 다이어그램에서 다양한 전이에 대응되는 데이터의 영향을 기술한 것으로 객체 흐름 다이어그램을 사용하는 시작점이다. 이것은 객체나 관계가 어떻게 생성되고 삭제되고 변형되는지 그리고 한 형태에서 다른 형태로 변하는 흐름에 대해서 설명한다.

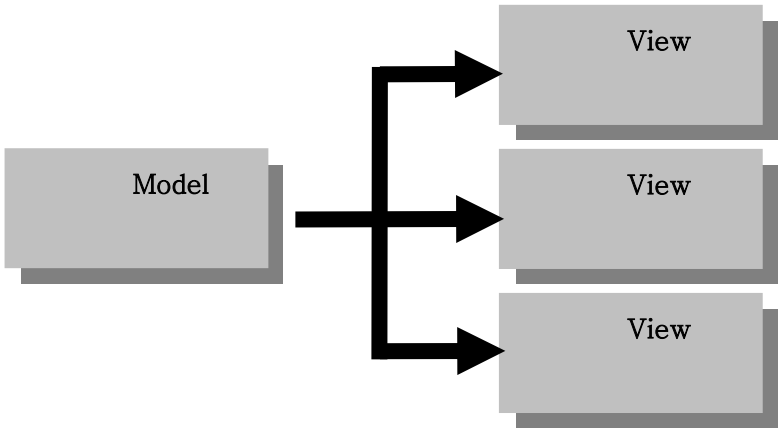
Transition	Effect on data
(call-design by ProjectManager) design act-design	Insert instance of modifies relationship with status:= passive
(call-review by ProjectManager) review act-review	Insert instance of reviews relationship with status := passive
call-create_version create_version act-create_version	Nothing: effect is postpond until after either call-create_first or call-create_next
call-create_first create_first (act-create_first ; i.m.)	Insert DesignDocument with versionnumber:= 1
call-create_next create_next act-create_next	Insert DesignDocument with right version number
call-open_for_modif. open_for_modif. (act-open_for_modif. ; i.m.)	Update modifies.status := active
call-modify modify (act-modify ; i.m.)	Update DesignDocument.content := <interactive input>
call-close_modification close_modification (act-close_modification; i.m.)	Update modifies.status := passive
call-open_for_review open-for_review (act-open_for_review; i.m.)	Update reviews.status := active
call-close_review_not_OK close-review_not_OK act-close_review_not_OK	Update reviews.status := passive
call-close_review_OK close-review_OK act-close_review_OK	Delete instance of reviews

[그림 2.2.15 : DFD에서 다양한 전이에 대응되는 데이터의 영향을 기술한 예]

2.2.3 도구

2.2.3.1 도구의 구조

SOCCA는 Leiden 대학에서 개발된 프로세스 모델링 방법론이다. 같은 대학에서 SOCCA 편집기(editor)도 개발하였다. 이 도구는 자바로 개발되었으며 웹 기반 도구로 사용되기 위해서 자바를 선택하였다고 한다. 이 편집기는 데이터 관점만을 지원하며 향 후 다른 관점들도 추가할 예정이라고 한다. SOCCA의 특징인 다중 뷰(view)를 지원하기 위해서 MVC(Model View Controller)를 기본 구조로 하였다. 그러나, Controller가 Model과 View와 강한 의존성을 가지므로 여기서는 Controller를 분리하지 않고 Model과 View에 포함되도록 하였다.

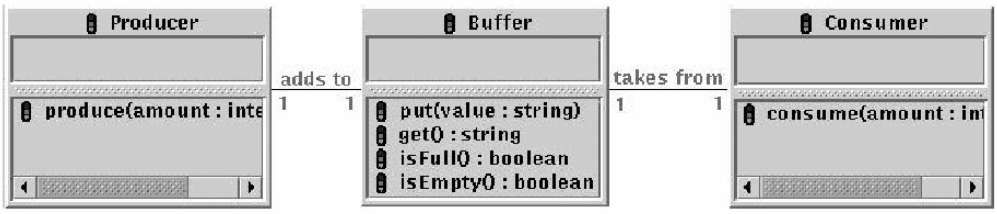


[그림 2.2.16 : SOCCA 편집기에서의 모델과 뷰의 관계]

이 도구는 세 부분으로 구성되어 있다. 이 부분들은 하나의 패키지(package)에 포함된다. 모든 모델을 포함하는 모델 패키지가 있고 각각의 모델에 대응되는 뷰(view)들을 포함하는 뷰 패키지가 있다. 이 패키지는 또한 뷰 클래스가 사용자와 상호 작용할 때 사용되는 다이얼로그(dialog)도 포함한다.

2.2.3.2 예제

여기서는 생산자-소비자 패턴(Producer-Consumer Pattern)을 SOCCA 편집기로 모델링한 예를 보여준다. 생산자-소비자 패턴은 세 개의 클래스로 구성된다. 이것은 Buffer, Producer 그리고 Consumer이다. Buffer는 생산자가 만든 아이템들을 소비자가 소비할 때까지 가지고 있는다. Producer는 put(value: string)을 호출함으로써 Buffer에 아이템을 추가할 수 있다. Buffer의 크기가 제한이 있으므로, Producer는 isFull(): boolean이 true를 돌려주면 아이템을 추가할 수 없다. Consumer는 isEmpty(): boolean이 true를 돌려주지 않는 한 get(): string을 호출하여 아이템을 한 개씩 Buffer에서 가져올 수 있다. Producer는 produce(amount: integer)를 호출하여 더 많은 아이템을 생산하도록 할 수 있다. Consumer는 consume(amount: integer)를 호출하여 아이템들을 소비한다. 이것을 SOCCA 편집기를 사용하여 그리면 다음과 같다.



[그림 2.2.17 : 생산자-소비자 패턴의 SOCCA 모델]

2.3 MERLIN : Supporting Cooperation in Software Development Through a Knowledge-Based Environment

2.3.1 개발배경 및 특징

MERLIN은 독일의 Dortmund대학에서 수행된 Merline project[1]를 통해 개발된 프로세스 중심의 소프트웨어 개발 환경의 프로토타입으로, 프로세스를 기술하고 실행시키도록 하는데 rule기반 기술을 사용하고 있다. MERLIN의 사용자는 크게 두 그룹으로 나뉘는데, 프로세스 엔지니어 그룹과 개발자 및 관리자 그룹이다. 먼저 프로세스 엔지니어 그룹의 경우는 MERLINE rule(prolog기반의 프로세스 모델링 언어)를 통해 특정 프로세스를 정의할 책임을 가지고 있고, 개발자 및 관리자 그룹의 경우는 정의된 프로세스 내부에 포함되어 소프트웨어를 생산하는 실제 업무에 대한 책임을 가지고 있다. 이 시스템이 제공하는 장점은 개발과 관리 행위들의 통합을 컴퓨터의 지원을 받아 이루었다는 점(the computer supported integration of development and management activities)이다. 그 예로 프로젝트 관리자가 현재 프로젝트 상태에 대한 온라인 정보를 얻을 수 있고, 이에 대해 필요한 행위를 즉시 개발자에게 알릴 수 있다.

2.3.2 MERLINE을 이용한 프로세스 정의와 실행

2.3.2.1 MERLIN의 프로세스에 대한 관점

MERLIN의 프로세스에 대한 관점은 다음의 정의된 엔터티들로 요약될 수 있다.

- 활동(Activity)
  소프트웨어 제품의 생산과 관련된 목표를 성취하기 위한 작업들의 집합
  (예: specify, edit, compile, test a module 등)
- 역할(Role)
  논리적으로 매우 밀접하게 관계되는 활동들의 그룹
  (예: project management, design, test 등)
- 문서(Document)
  소프트웨어 개발 프로세스 동안 생산되어질 수 있는 것들
  (예: modules, documentation, test plans 등)
- 자원(Resource)

[1] Merline은 소프트웨어 내부 블록들을 그들의 속성과 관계를 이용하여 잘 구성함으로써 소프트웨어를 개발할 수 있도록 하자는 아이디어를 담고 있다. 이러한 정보들로부터 Merline과 같은 프로세스 중심의 소프트웨어 개발 환경은 개발자들과 관리자들에게 최대한의 정보를 제공할 수 있고, 소프트웨어 프로세스 내에서 활동들을 수행할 수 있도록 지원할 수 있다. Merline이라는 이름은 영국의 “아서왕 이야기”에 나오는 마법사 Merline의 이름을 딴 것으로 미래를 직시하여 아서왕이 그의 나라를 잘 다스릴 수 있도록 계획을 세우는데 도움을 주었다.

소프트웨어 생산에 참여하는 사람들과 소프트웨어 개발활동을 지원하는 도구와 같은 기술적 자원들

(예: editors, debuggers 등)

사용자들은 한 개 이상의 역할과 관계되어지고, 문서는 특정 문서에 행해질 수 있는 행위들의 집합과 이들 행위들을 지원하는 도구의 집합에 제한되어진다.

2.3.2.2 MERLIN의 프로세스 정의 : PROLOG-based process definition language

MERLIN의 프로세스 엔진은 주어진 프로세스 정의를 실행하면서 사용자의 작업환경을 설정하고 변경하는 MERLIN 내에 있는 부분이다. 활동은 특정 활동을 수행할 수 있는 역할, 그 활동에 대해 책임이 있는 사람, 적절한 문서에 대해 관련된 접근권한, 그리고 다른 활동들과의 의존관계 등의 내용들을 갖는 선행조건들로 구성된다. 어떤 활동의 수행은 이에 대한 결과가 다른 활동에 대한 선행조건들을 만족시킬 수 있으므로 수행해야 할 새로운 활동을 알려주게 된다. 따라서 프로세스 엔진의 주요한 작업은 바로 이러한 행위의 선행조건들을 평가하는 것 뿐만 아니라 그 결과에 따라 적절하게 작업환경을 사용자에게 변화시켜 보여주는 것이 된다. 덧붙여 프로세스 엔진은 작업환경을 보여주는 것 뿐만 아니라 현재와 이전 프로세스 상태들에 대한 설명을 가능하게 한다. 예로 어떤 환경은 다음과 같은 사용자의 질문에 대한 답을 제공할 수 있다. : 왜 내가 모듈 m1_c를 코딩해야 하는가?, 누가 이 프로젝트에 투입되었는가?, 누가 모듈 m1_c의 스펙을 변경시켰는가?, m1_c의 코딩은 언제까지 해야 하는가?. 이러한 질문들은 MERLIN에서 제공하는 정보버튼을 통해 사용자가 제시할 수 있다. 그리고 프로젝트 수행 동안 중요한 요구사항으로 인해 프로세스가 변화할 수도 있는데, 이러한 일들은 자주 발생된다. 프로세스는 그 일부가 진행도중 결정되는 사항들에 의존적이므로 미래의 상황들에 대해 전체적으로 결정지을 수 없는 특징을 가지고 있다. 결국 소프트웨어 프로세스를 정의하는데 사용될 언어는 명확하게 정의되어진 어의(semantic)을 가져야 하고, 프로그램이 효율적으로 해석되어질 수 있을 정도로 충분하게 간단해야만 한다.

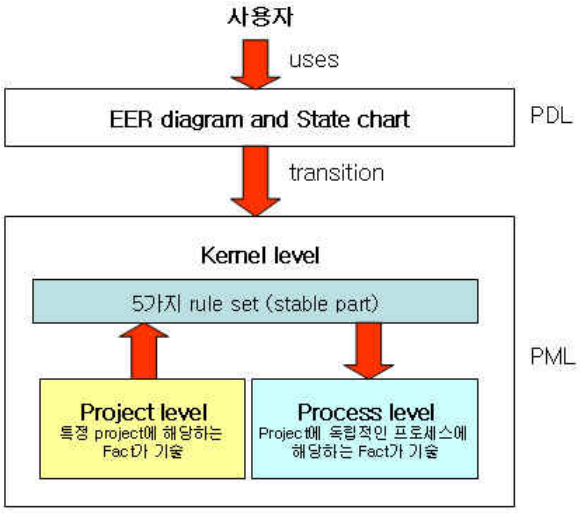
이러한 이유로 MERLIN에서는 프로세스 정의 언어(Process Definition Language)로서 PROLOG기반의 rule기반 언어를 선택하였다. 그리고 작업환경의 구축 또는 현재와 이전 프로세스 상태에 대한 질문들은 유일하게 PROLOG goal로 정의되어진다. 앞으로 다음과 같은 시나리오를 예제로 하여 MERLIN rule에 대해 설명할 것이다.

Smith와 Miller라는 두 명의 개발자가 있다. Smith는 모듈의 테스팅에 대한 책임을 가지고 있는 quality assurance engineer라는 역할을, 그리고 Miller는 모듈의 코딩과 리뷰에 대한 책임을 가지고 있는 programmer라는 역할을 담당하고 있다. 프로세스는 다음과 같다.

먼저 programmer에 의해 코딩 및 리뷰를 거치고 간단히 테스팅되어진 모듈을 quality assurance engineer가 test plan에 기반하여 보다 확장된 테스트를 수행하게 된다.

[그림 2.3.1 : 예제 시나리오]

소프트웨어 개발 프로세스는 Kernel, Process, Project 레벨의 세 가지 레벨로 나뉘어 기술된다. 여기서는 그림 3.2의 PML(Process Manipulation Language) 측면에서 살펴보도록 하고, PDL(Process Design Language) 측면에서는 다음 절에서 설명하도록 한다.



[그림 2.3.2 : MERLIN의 프로세스 기술 언어 개요]

먼저 **Project 레벨** 에서는 특별한 프로젝트 상태가 정의된 미리 정의되어진 predicate의 set을 이용하여 기술한다. 예를 들어, 프로젝트의 이름, 프로젝트에 참여하는 사람들의 이름과 역할, 그리고 그들의 책임 등이 기술된다. 그림3은 이에 대해 기술되어진 rule의 예로 5개의 fact를 통해 프로젝트 상태의 일부를 정의하고 있다.

```
project(merlin_demo_project, [smith, miller]).
has_roles(miller, [programmer]).
has_roles(smith, [quality_assurance_engineer]).
responsibilities(miller, programmer, [m1_c]).
responsibilities(smith, quality_assurance_engineer, [m2_tp]).
```

[그림 2.3.3 : Project 레벨에서 사용된 rule의 예]

Process 레벨 에서도 역시 미리 정의되어진 predicate들의 set을 이용하여 기술한다. 이 레벨에서는 특정 프로젝트와는 별개의 독립적인 프로세스의 구성요소들을 정의하게 된다. 이 구성요소에는 document의 타입, 가능한 document의 state와 state transition 들을 포함한다. Fact의 한 예로, document_type_states는 특별한 document 타입의 가능한 모든 state들을 정의할 수 있다. 이것의 예로, specification이라는 document 타입과 c_module의 상태들의 정의는 그림 3.4와 같이 fact로 기술된다.

```
document_type_states(specification, [incomplete, not_yet_designed, designed, complete]).
document_type_states(c_module, [incomplete, not_yet_implemented, implemented,
not_yet_compiled, compiled, not_yet_tested, complete])
```

[그림 2.3.4 : Process 레벨에서 사용된 rule의 예]

이것 이외의 몇 개의 fact의 예를 들면 다음과 같다.

1) document_type_tools
어떤 도구가 어떤 상태와 접근권한을 가져야 사용될 수 있는지를 정의하는 fact
이 fact를 통해 사용자의 화면에 보이는 작업환경 내의 document에 행해져야 할 메뉴 리스트를 결정할 수 있다.
예) document_type_tools(c_module, ascii_pager, readable, Any_state, []).
: c_module을 readable권한으로 접근할 수 있으면 ascii_pager 도구가 실행된다.

2) document_relation_type
document타입들간의 관계 타입을 기술하는 fact
이 fact를 통해 document들 간의 context-sensitive한 관계들을 저장할 수 있다. 이것은 특정 document와 관련된 document들의 상태가 변경되었을 때, 사용자들의 작업환경에 반영하기 위해 필요하다
예) document_relation_type(is_implemented_in, specification, c_module).
: Specification과 c_module과의 관계는 is_implemented_in이다.

3) consistency_condition와 automation_condition

document의 상태변화는 consistency condition과 automation condition에 의해 전환되어진다. 이들은 사용자나 batch tool이 특정 document의 상태를 변경시켰을 때 관련된 document들의 상태가 어떻게 변화되었는지를 기술한다.

Consistency_condition fact는 어떤 document의 상태변화가 다른 document들의 상태변화를 요구할 때 process의 consistency을 유지하기 위해 사용되어진다.

예) consistency_condition(c_module, Any_state, incomplete, [[source, is_implemented_in, not_yet_designed],[source, realization_is_imported_in, not_yet_designed]]).

: 만약 is_implemented_in관계에 있는 원본 document의 상태가 not_yet_designed상태이거나, realization_is_imported_in 관계에 있는 원본 document의 상태가 not_yet_designed상태에 있는 경우, c_module은 현재 상태와는 독립적으로 incomplete상태로 된다. 이것은 관련된 specification들 중 하나가 변화되었다면 c_module은 더 이상 접근 할 수 없다는 것을 의미한다.

Automation condition fact는 선행한 활동이 올바르게 완료된 경우 수행해야 할 새로운 활동들을 수행할 수 있도록 document 상태를 변경한다.

예) automation_condition(c_module, incomplete, not_yet_implemented, [[source, is_implemented_in, complete], [source, realization_is_imported_in, complete]]).

: c_module과 is_implemented_in관계에 있거나 realization_is_imported_in관계에 있는 원본 document들의 상태가 complete가 될 경우, c_module의 상태가 incomplete에서 not_yet_implemented 상태로 자동으로 변경된다.

4) roletype_document_work_on

role name과 함께, 이 role과 관련된 document와 접근권한 등을 기술한다.

예) roletype_document_work_on(programmer, c_module, not_yet_implemented, [[specification, complete], [c_error_report, with_errors], [review, review_rejected]]).

: c_module의 상태가 not_yet_implemented인 경우, programmer의 작업환경에는 c_module이 보이게 된다. 이 경우, 관련된 specification이 compelete상태라면 specification은 readable 접근권한을 가지고 작업환경에 보여지게 되고, c_error_report가 with_errors 상태라면 이것 역시 작업환경에서 readable 접근권한을 가지고 작업환경에 보여지게 된다. 게다가 만약 이것이 review문서가 reject_reviewed 상태라면 관련된 review 문서가 작업환경에 보이게 된다.

마지막 **Kernel 레벨**의 프로세스 정의는 위에서 정의한 fact들을 기반으로 사용하여 PROLOG-like rule들의 set에 의해 정의되어진다. 이 레벨은 MERLIN의 ProcessEngine

과 WorkBench 창들과 내용들을 관리하고, 작업환경 정보를 구성하고 변경하며, 프로세스 기술(description)에 있어 가장 안정된 부분으로 사용자 인터페이스에 대한 패러다임이나 형상관리 등의 새로운 요소가 추가될 때에만 변경된다. 그리고 소프트웨어 프로세스에 대한 변화는 Kernel에 대한 변화를 요구하는 것이 아니고 프로세스와 프로젝트에 대해 기술하는 fact들에 대한 변화만을 요구한다. Kernel rule set은 다음과 같은 다섯가지의 소위 rule cluster들로 구성되어지는데, 이것은 StartUp, WorkingContext, ChangedStates, TransactionManager, LockManager 이다.

1) StartUp

사용자 로그인, ProcessEngine창의 project, role, activity, 그리고 정보선택을 관리하는 모든 rule들을 포함하고 있다.

2) WorkingContext

사용자의 작업환경에 보이기 위해 필요한 정보를 선택하기 위한 모든 rule들을 포함하고 있다.

예) working_context(Ident, Project, Role, Menu_Activity_List, Document_List, Relation_List) :-

```
responsibilities(Ident, Role, Responsibility_List),
documents(Role, Responsibility_List, Document_List),
...
documents(Role, Responsibility_List, Document_List) :-
...
document_state(Doc, State),
roletype_document_work_on(Role, Type, State, Rdoctypes),
...
```

: Precondition은 Ident로 표현되는 사용자가 Project에서 Role을 가진 경우 프로젝트 내의 responsibility를 갖는다는 것을 나타낸다. 덧붙여, 사용자의 responsibility가 정의되어지고 난 후에는 관련된 document들은 허용된 state를 갖도록 해야 한다.

3) ChangedStates

document들의 상태가 변경되었을 때 관련된 프로세스 정보를 갱신하기 위한 rule들을 포함하고 있다. 그리고 envelope(interactive tool, batch tool들)에 의해 변화될 document들의 미래 상태들을 예측하기 위한 목표를 정의한다. 첫번째와 두번째 rule에서는 consistency나 automation condition에 따라 상태변화를 정의하고, 세번째 rule에서는 batch tool들의 invocation을 정의한다. 그리고 마지막 rule은 관련된 document들의 상태변화를 정의한다.

예) changed_states(Ident, Project, Role, Document, State) :-

```
document(Document, Type, Path),
consistency_condition(Type, State, New_state, Condition_list),
...
changed_state(Ident, Project, Role, Document, New_state).

changed_states(Ident, Project, Role, Document, State) :-
document(Document, Type, Path),
automation_condition(Type, State, New_state, condition_list),
...
changed_states(Ident, Project, Role, Document, New_state).

changed_states(Ident, Project, Role, Document, State):-
document(Document, Type, Path),
document_type_tools(Type, Call, Access, State, [New_state_failure,
New_state_success]),
document_all(Document, Call, Program_to_call),
CALL(program_call, Document, Program_to_call, New_state_failure,
New_state_success, New_state),
REMOVE(document_state(Document, Old_state)),
INSERT(document_state(Document, New_state)),
document_state(Document, New_state),
changed_state(Ident, Project, Role, Document, New_state).

changed_states(Ident, Project, Role, Document, State) :-
related_documents(Ident, Project, Role, Document, States).
```

: 첫번째 rule은 현재 document에 대해 consistency condition을 확인한다. 만약 consistency condition이 존재하면, document의 상태는 변화하고, 만약 존재하지 않으면 두번째 rule이 automation condition을 확인한다. 만약 automation condition이 존재하면 document의 상태가 변화된다. 세번째 rule에서는 batch tool의 invocation이 점검되어지고, 현재 document의 모든 조건들이 점검되면 마지막 rule이 관련된 모든 document의 상태를 점검하게 된다.

4) TransactionManager와 LockManager

분산환경에서 동일한 document에 대해 여러 명의 사용자들이 접근하여 작업하는 경우

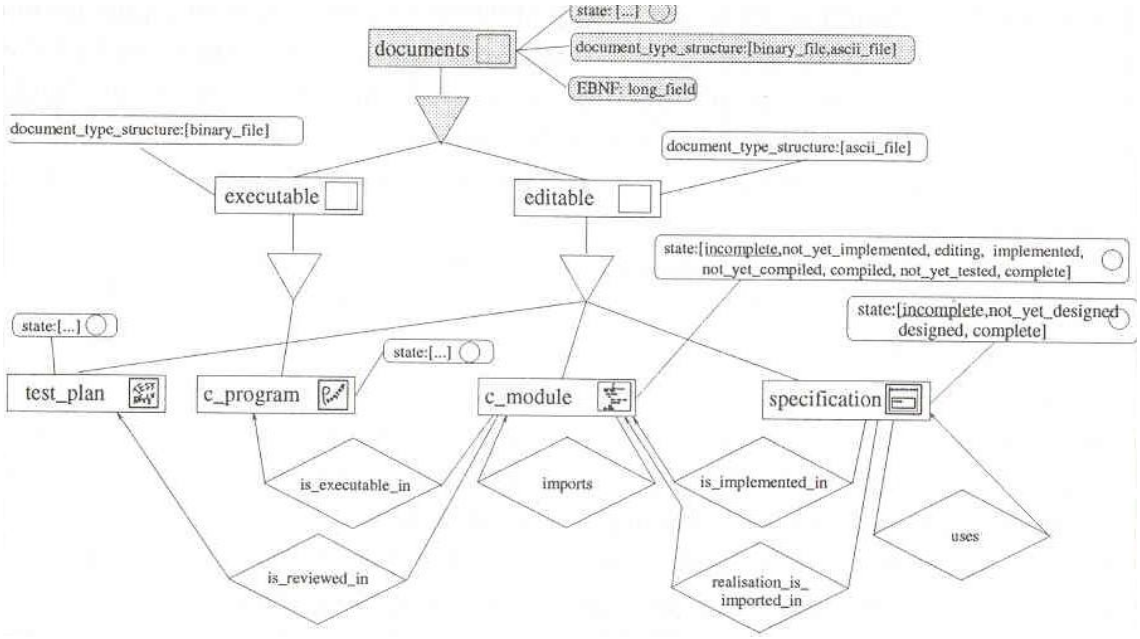
많은 문제점들이 발생할 수 있다. 동시에 작업하는 경우 read/write 또는 write/write와 같은 충돌이 일어나게 되고, 특히 ProcessEngine의 경우 batch job이 존재하므로 이것으로 인한 부가적인 충돌도 존재한다. 따라서 이러한 충돌을 해결하기 위해 MERLIN에서는 transaction mechanism을 제공한다. 이런 문제들에 대한 연구는 기존에도 존재했으나 각각의 장단점이 존재한다. 여기서는 PSEE에 이러한 해결방안들을 어떻게 적용할 것이며, 동시제어 전략을 결정하는데 프로세스에 대한 지식을 어떻게 사용할 것인지가 이슈가 된다. MERLIN에서는 사용자 작업환경 기반의 user interaction model을 적용한 transaction model을 개발하였다. 5개의 서로 다른 transaction type이 제공되며, 이는 사용자에 의해, 또는 ProcessEngine에 의해 trigger된다. 세부내용은 참고문헌[1]을 참고하도록 한다.

2.3.2.3 MERLIN의 프로세스 디자인 : Process modeling language

앞 절에서는 PROLOG기반의 PDL을 사용하여 소프트웨어 프로세스를 어떻게 기술하는가에 대해 살펴보았다. 이 장에서는 프로세스를 정의하기 위한 notation으로 보다 사용자와 친숙한 graphic notation에 대해 소개하고, 앞 절에서 소개한 시나리오를 통한 예제를 보이고자 한다. 프로세스 디자인을 위해 선택한 것은 EER(Extended-Entity-Relationship) diagram과 State chart이다.

1) EER diagram

이는 소프트웨어 프로세스의 static 영역만을 기술하는데 사용한다. 즉, △document와 그것의 attribute, △ relationship-type, △ tool-classes, △ roles이 된다. 그림3.5는 document type과 relationship type을 기술한 EER diagram을 보여준다. EER diagram은 모델링 개념으로 entity, attribute, relationship을 제공하고, 구조적인 표현을 위해 inheritance, refinement 개념을 제공한다.



[그림 2.3.5 : EER diagram의 예]

그림 3.5에서 음영이 있는 상자는 변하지 않는 부분이고 그렇지 않은 것들은 process engineer에 의해 프로세스 디자인 단계에서 변경될 수 있다. Process engineer에 의해 개발되어진 부분들은 abstract document type과 concrete document type으로 나뉘어지는데, 이것은 box로 표시되어지고, attribute 들은 타원으로 표시되어진다. 여기서 attribute는 서로 다른 4가지 type으로 나뉘는데, △ document들의 가능한 상태들을 기술하는데 사용되는 것과 프로세스의 dynamic한 부분을 기술하는데 사용되는 것들(타원으로 표시된 것들), △ 작업환경 내의 document들의 아이콘처럼 보이도록 기술되는데 사용되는 것들(box안에 아이콘으로 보이는 것들), △ document의 abstract syntax를 기술하는데 사용되는 것들(type처럼 긴 field를 갖는 것들), △ type enumeration의 속성들을 기술하는데 사용되는 것들(값들을 포함하는 대괄호들을 가지고 있는 것들) 이다. Inheritance를 이용해서 이미 정의되어진 attribute들은 프로세스에 있는 concrete document type들 각각에 사용될 수 있다. 위의 예에서 사용되는 concrete document type들은 test_plan, c_program, c_module, specification이고, attribute들은 state, document_type_structure, EBNF 가 있다.

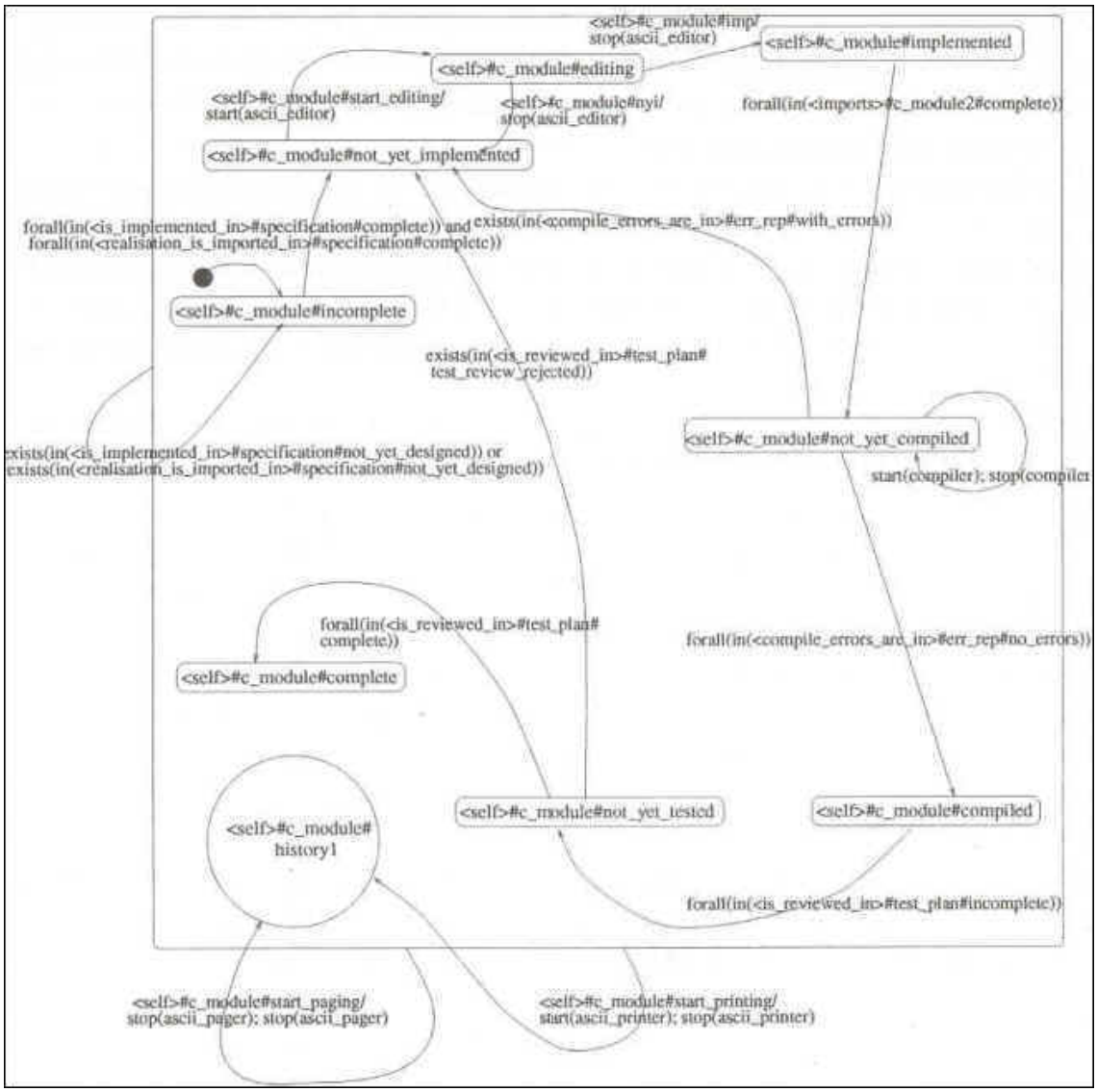
Relationship type들은 다이아몬드 형태로 나타내어지고 document type들 간의 가능한 관계들을 기술한다. 앞 절에서 살펴보았던 document_type_states은 attribute state와 대응되고, document_relation_type은 주어진 relationship과 대응된다.

2) State chart

이는 프로세스의 dynamic part를 기술하는데 사용된다. State chart는 state transition

diagram을 기반으로 하여 orthogonality, depth, 그리고 broadcast communication mechanism 부분을 개선한 형태이다. 이것은 state와 labeled transition개념을 제공하고 있으며, label은 dynamic한 행위를 기술하는데 사용되는 event/action의 쌍으로 이루어진다. 지원되는 broadcast mechanism은 모든 내/외부 event를 sender와는 독립적으로 system의 모든 부분에서 받을 수 있다는 것을 보장한다.

그림 3.6에서는 c_module type의 document의 dynamic 행위를 기술하는 state chart를 보이고 있고, 여기서 한 예로 module의 상태가 implemented에서 not_yet_compiled로 변경되는 것을 기술하는 dynamic한 제약사항을 고려해보자. 이것은 state chart에서 해당 상태들 간의 labeled transition으로 기술되어진다. Label은 imports relationship에 의해 관계된 document들과 관련있는 c_module type을 갖는 모든 document들이 complete 상태에 있다면 상태변경이 발생한다는 것을 기술하고 있다.

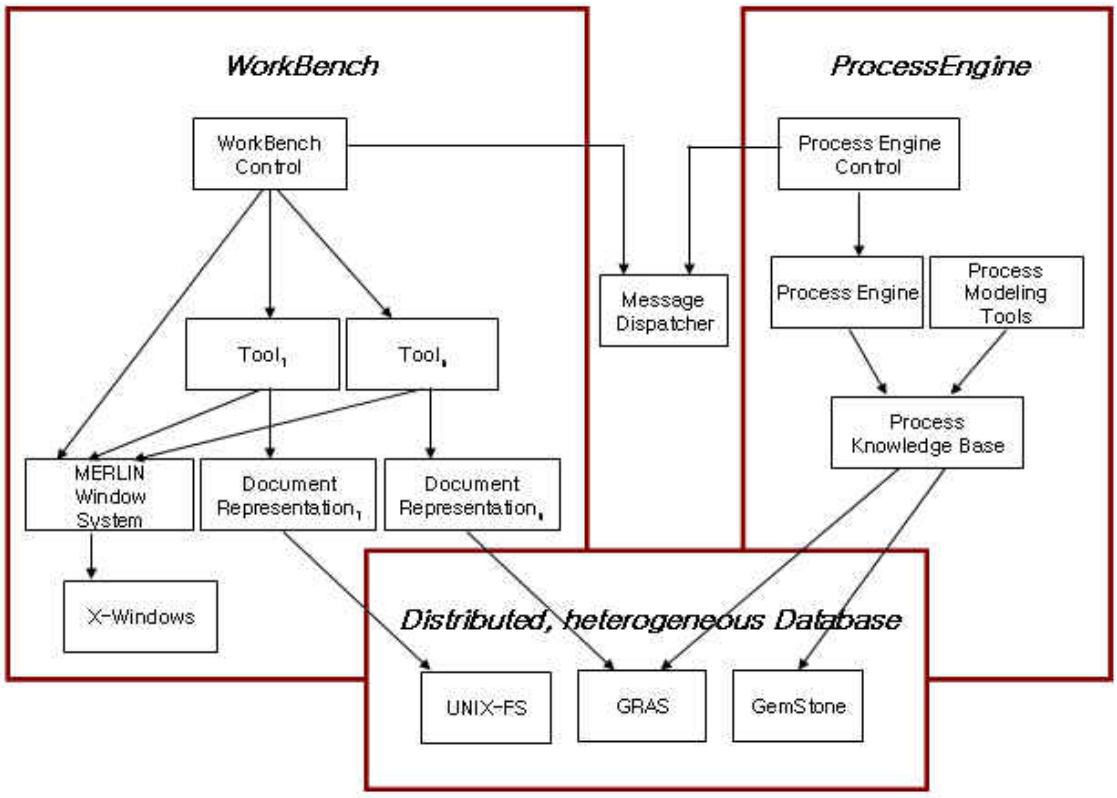


[그림 2.3.6 : State chart의 예]

2.3.3 도구

2.3.3.1 아키텍쳐

그림 3.7은 MERLIN의 architecture의 전반을 보여주고 있다. 이 그림에서 box는 module이나 subsystem을 나타내고, 화살표는 use-relationship을 나타낸다. 아키텍쳐는 각 ProcessEngine을 가지고 있는 여러 개의 WorkBench들로 이루어져있는 분산모델을 기반으로 하고 있다. 그리고 이것은 단지 하나의 ProcessEngine과 WorkBench의 쌍을 보여준다. ProcessEngine과 WorkBench는 TCP/IP를 기반으로 하는 Message Dispatcher를 통해 communicate한다. Process Engine Control은 Process Engine에서 산출된 작업환경을 WorkBench에 보낸다. 그리고 WorkBench Control은 관련된 screen display의 초기화 및 변경된 상태정보를 보이며, 이를 ProcessEngine에게 결과를 알린다. 프로세스 데이터는 그림 하단의 데이터베이스에 저장이 되는데, 이것은 분산된 방법으로 저장되는 것은 아니다. 단지 process interpreter가 여러 인스턴스에 존재할 뿐이다. Message Dispatcher와 관련해서 MERLIN은 HP Softbench를 사용했다.



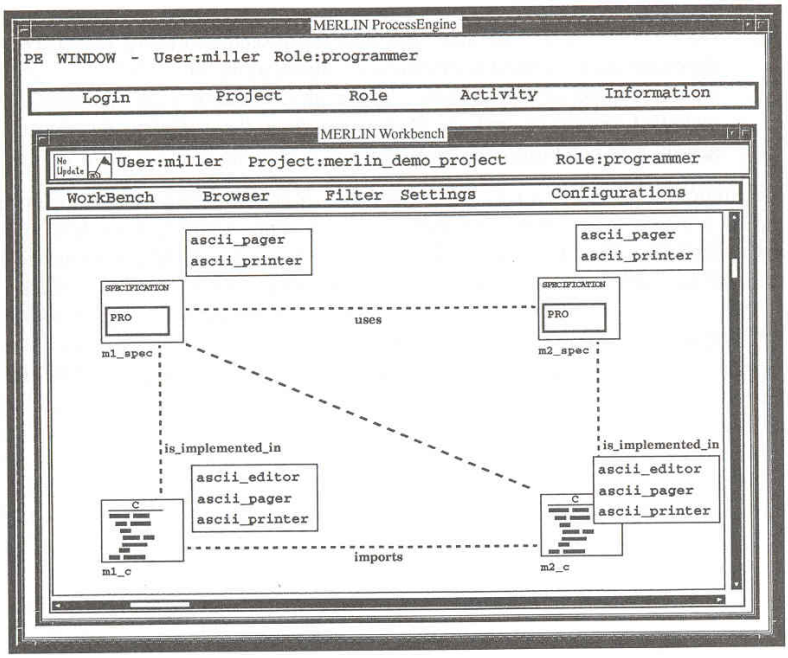
[그림 2.3.7 : MERLIN의 아키텍쳐]

Process Engine은 PROLOG-rule처럼 기술되어진 개발 프로세스를 저장하고 추출하고 해

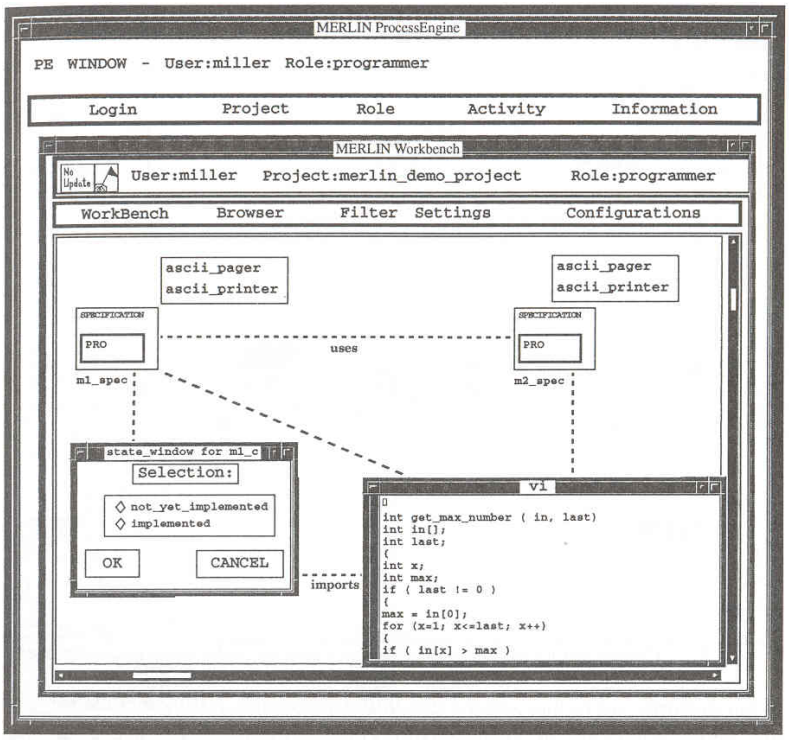
석하는 역할을 담당한다. 특히 transaction synchronization이 구현되어 있다. Process modeling tool은 EER diagram과 state chart를 이용하여 프로세스를 기술하기 위한 edit와 analysis 기능을 제공한다. 그림 3.7 하단에 있는 프로세스 정보와 document를 저장하기 위한 DBMS는 MERLIN과 heterogeneous하다. 몇몇의 document는 UNIX file system에 저장이 되고, 용량이 큰 경우는 GRAS라고 하는 데이터베이스에 ASG(Abstract Syntax Graph) 형태로 저장이 된다. 특히 Kernel, Project, Process 레벨의 프로세스 데이터 중 rule들은 GRAS에, fact들은 GemStone이라는 객체지향 데이터베이스에 나뉘어 저장된다. GRAS와 GemStone의 경우 rule과 fact에 대한 granularity문제 및 locking문제를 해결한다.

2.3.3.2 사용자의 작업환경(working context)

MERLIN은 사용자들에게 그들의 현재 role과 관련된 "작업환경(working context)"을 제공하고, 이를 통해 작업환경에서는 구현해야 할 document들, 그리고 이 document들과 다른 document들간의 의존관계, 그리고 각 document들에 수행될 수 있는 activity등 관련된 정보들을 보여준다. 그림 3.8, 3.9에서는 2.3.2.2에서 제시되어진 시나리오를 바탕으로 한 programmer role을 가진 Miller의 작업환경을 보이고 있다.
여기서 document는 box로 표현이 되고, 이 box는 관련된 document에서 행해져야 할 행위들과 관련된 context-sensitive 메뉴들을 가지고 있다. 그리고 box들간의 레이블이 있는 화살표는 document들간의 관계를 나타낸다. Box 내의 context-sensitive 메뉴를 선택하면 관련된 activity와 도구가 실행된다.



[그림 2.3.8 : 작업환경의 예]



[그림 2.3.9 : 작업환경에서 context-sensitive menu 및 tool invocation 예]

2.4 OIKOS : Constructing Process-Centered SDEs

2.4.1 개발배경 및 특징

OIKOS는 1992년부터 ESPRIT BRA Promoter working group에 의해서, Italian Ministry for university and Research의 fund를 지원 받아서 시작되었다. OIKOS는 house를 의미하는 고대 그리스어로서, 현재 과학계에서는 environment를 나타내는 어근으로 사용된다. 따라서, OIKOS는 software development의 environment를 의미하며, OIKOS project의 목적은 process-centered software development environment의 construction을 쉽게 해주는 것에 있다. 구체적으로, OIKOS가 제공하는 기능은 크게 3가지로 나눌 수 있다. 첫째로, 모든 관련된 perspective들을 고려한 software development process를 modeling하는데 필요한 concept과 notation을 제공한다. 둘째로, 서로 다른 level의 abstraction에서 process를 modeling함으로써 process에서 각각의 role을 하는 actor들에게 자연스러운 process의 personalized view를 제공한다. 마지막으로, model의 enactable level까지 refinement를 지원함으로써, off-the-shelf tool들의 import와 development environment의 distribution을 쉽게 할 수 있다.

2.4.2 Modeling Concepts, Methods and Languages

2.4.2.1 OIKOS의 Modeling Concepts

OIKOS는 다음 4가지의 principle에 바탕을 두고 있다.
첫째, enactable software process model은 interacting reactive system이 hierarchy를 통해서 구성되어 있다.
둘째. Model의 system은 entity라고 불리며, 다른 class들에 속한다. 각 class들은 operating environment, manage site, actor's role과 같은 modeling concept들을 구현한다.
셋째, Customizable predefined service는 allocated resource들에 access를 해서, process enactment를 지원하는 기본적인 functionality를 제공한다.
넷째, enactable model은 step-wise refinement를 통해서 개발된다.

Class의 entity는 simple이거나 structured한 속성을 가지고 있다.
Simple entity는 structure를 가지고 있지 않고, model의 hierarchical 정의에서 leave에 해당한다. Structured entity는 internal node로서 그들의 구조는 hierarchy를 정의한다. Structured entity의 목적은 그들의 part들을 coordinate함으로써 전체 process에 공헌을

하는 것이다.

	process	Office	environment	desk	cluster	session	role	Service instance	coordinator
process	X	X	X		X	X		X	X
office		X	X		X	X		X	X
environment				X	X	X		X	X
desk					X		X		X
cluster	X	X	X		X	X			X

[표 2.4.1: Entity Structure]

표 2.4.1의 열은 internal node의 모든 entity class들을 나타내고, 행은 structured class들을 나타낸다. X 표시는 열에 있는 class의 entity가 행에 있는 class의 entity의 part가 될 수 있다는 것을 의미한다.

다음으로 복잡도가 증가하는 순서대로 각 class들과 연관된 modeling concept들을 소개한다.

● Coordinator

이름이 의미하는 대로, 이 class의 entity는 그들의 parent의 다른 part와 coordinate하게 한다. 또, 다른 entity로 stimuli를 보낼 수 있다. 이를 통해서, entity들이 message passing을 통해서 서로 영향을 줄 수 있게 된다.

● Service Instance

어떤 concrete resource를 instance가 관리할 수 있는지 명시하는 역할을 한다. 그리고, resource에 대한 access 시 concurrency control에 관련된 미리 정의된 policy를 customize할 수 있다.

● Session

Service instance에 의해서 관리되고 있는 resource에 대한 access point를 정의한다. 또, interaction protocol을 명시한다.

● Role

Software process에서 활동하는 actor들의 action을 modeling하는 entity이다. 각 role은 enactment중의 actor의 display window에 해당한다.

● Desk

Cooperative role을 수행하는 actor들의 group이 공유하는 job의 state에 대한 지식과, 이 state들이 어떻게 evolve되는지에 대한 지식을 정의한다. Desk는 지역적으로 working document에 대한 state 정보를 가지고 있어서, actor가 무엇을 다음에 해야 할지 결정하는데 있어서, desk의 view를 통해서 결정을 도와준다.

● Environment

Actor가 workplace(computer power를 공유하며, working document와 적용되는 tool들이 임시 저장된다.)에 access하는 것을 조정하고 지원한다. 또, 작업하는 document를 꺼내주거나 저장해주어서 actor가 desk를 통해서 작업할 수 있게 해준다.
Tool들을 install하게 해주고, 적용 가능하게 해주며, 관련된 actor들의 access right을 조절하여, model에 순응하게 하는 역할을 한다.

● Office

여러 environment들간의 coordinate하는 역할을 한다.

● Process

Development process 전체이던지, 또는 일부를 modeling한다. 특히 OIKOS에서는 managed activity를 구성하는 것을 강조하고 있다. 또, activity를 시작시키는 외부의 stimuli를 명시한다. 그리고, process는 resource를 소유하여, process와 관련된 document, 개발 tool들을 저장할 수 있다.

● Cluster

Structured entity를 group화한 abstract description을 지원함으로써, modularity를 향상시킨다.

2.4.2.2 OIKOS의 모델링 기법

Entity를 기술하기 위해서, abstract와 concrete view를 각각 사용한다. OIKOS는 모델을 구성하는데 있어서 top-down refinement method를 지원하여, abstract view가 먼저 소개되고, 다음으로 concrete view로 refine된 후, 마지막으로 enactable하게 변환되게 된다.
Abstract view는 entity가 반응하기 위해 필요한 stimuli를 제거하여, 단지 entity간의 관계를 추상화 한다. Refinement를 통해서, 각 part들이 필요한 stimuli들을 고려해서 좀 더 복잡해지게 된다.

모든 entity들이 concrete view를 가지고 있다고 해도, 그 model이 enactable한 것은 아니다. OIKOS는 어떤 predicate들은 enactment전까지 모호한(oracular) 상태로 남아있게 허용해 준다.

2.4.2.3 OIKOS의 모델링 언어

OIKOS에서는 specification language인 Limbo와, enactment language인 Pate를 사용한다. Pate는 Limbo의 sub-language로서, Extended Shared Prolog(ESP)에서 유도된 concurrent logic language이다.

두 가지 언어가 공통적으로 가지고 있는 특징은 다음과 같다.

- Entity들은 tree로 구성된다.
- Entity의 coordinator는 blackboard model에 따르는 concurrent agent이다.
- Coordinator는 pattern이라 불리는 reaction rule들의 집합이다.
- 여러 개의 조건들이 만족되면, 그 중 하나가 nondeterministic하게 선택된다.

Pate는 coordinator를 정의할 때, path expression을 추가할 수 있다. Motif기반의 interface를 제공하고, run-time support인 Expo를 가지고 있다.
Expo는 off-the-shelf tool들의 import, Pate system들의 분산의 조절, standard service 등을 지원해준다.

Limbo는 다음과 같이 Pate를 extend 하였다.

- Entity의 state가 intentional description도 허용한다.
- Description에 사용되는 몇몇 predicate들은 oracular로 선언될 수 있다.
- Structured entity의 concrete description은 그 entity의 부분에 대한 abstract view로 나타내진다.

Limbo와 Pate는 specification과 분산 시스템 구현에 사용되는 general purpose 언어이지만, OIKOS에서는 software process의 enactment에 특성화가 되어 있다.

2.4.3 System Architecture and tools

Expo는 OIKOS run-time support이다. Expo의 주목적은 Pate system의 분산 execution 지원과, off-the-shelf tool들과 non-Pate system들과의 통합, Pate system과의 다중 human interaction이다.

이 목적을 이루기 위해서, 몇 가지 tool들이 개발되었다.
Compiler와 intermediate code interpreter, distribution control monitor, resource allocation monitor, user interface등이 그 tool들이다.
Compiler는 Lex/Yacc를 사용해서 개발했으며, program을 intermediate code로 변환시킨다. Interpreter는 Prolog의 일종인 BIMprolog로 작성되었으며, 다른 언어로 작성된 module을 integration시켜준다. User-interface는 OSF/Motif 기반으로 되어 있으며, monitor는 Pate를 사용해서 구현되었다. 이 모든 부분들을 통합시켜주는 glue는 C 언어이고, process간의 통신은 socket을 통해서 이루어진다.

Expo는 서로 message를 교환하는 component들의 집합으로 이루어진 open system이다. Component는 Pate system과 non-Pate system으로 나누어지고, non-Pate system은 다시 stand-by와 load-and-go로 나누어 진다. Stand-by component는 오랜 시간 존속하여서, 요청을 받고 답을 보내는 것이 가능하나, load-and-go component는 짧은 시간 동안 존속하여서, 주어진 data에 대해 single computation만 수행한다.

Expo는 3가지의 process에 의존하고 있다.

(1) Network communication server : component들간의 point-to-point protocol을 구현한다.
(2) Expo name server: Expo component들의 space address를 유지한다.
(3) Activator: 각 component를 implement하는 각 process들의 활성화와 종료를 조절한다.

2.5 ALF : A Framework for Building Process-Centered Software Engineering Environments

2.5.1 개발배경 및 특징

ALF 프로젝트는 1987년에서 1992년에 걸쳐 ESPRIT 프로그램의 Commission of the European Communities의 후원으로 진행된 것으로서 프로젝트명은 "Advanced Software Engineering Environment Logistics Framwork/Accueil de Logiciel Futur"이다. ALF는 소프트웨어 개발을 지원하기 위한 컴퓨터화된 도구를 제공하는 것을 목적으로 한 연구 프로젝트로서 ESPRIT의 PCTE[2](Portable Common Tool Environment) 프로젝트의 결과를 활용하고 지식기반 시스템(Knowledge Based System)과 정보 시스템(Information Systems) 기술을 통합 적용하여 좀 더 사용자의 활동과 목적에 부합할 수 있는 SEE를 개발하고자 하였다.

ALF는 소프트웨어 프로세스 모델(Software Process Model)을 환경과 통합한 것인데, 이것은 이 ALF PSEE가 실행하여야 하는 활동들(Activities)에 대해서 알려주고, 이러한 활동들을 수행해야 하는 사람들을 선언하고, 활동들을 통제하는(활동들의 시작 또는 종료 컨디션) 정책들을 정의하는 등의 활동을 포함하고 있다. 이러한 특징들을 위해서는 Process Modeling concept과 Process Model Description Formalism, 그리고 프로세스 모델의 묘사를 관리할 수 있는 정확한 메커니즘을 필요로 하고 또한 묘사된 프로세스 모델에 따라 소프트웨어 프로세스를 진행하기 위한 Process Enactment Mechanism의 정의와 구현이 필요하다. ALF 프로젝트에서 설정한 목표는 다음과 같다.

1) 사용자의 주도를 통제하고 정의된 프로세스 모델에 어긋나는 활동을 제한한다.
2) 시스템이 적극적인 역할을 하도록 한다.
   - 사용자 입장에서 Constructive Initiative를 하도록 한다.
   - 정의된 프로세스를 벗어날 때는 프로세스 일관성을 위해 Corrective Initiative를 취한다.
3) 다음과 같은 기능을 제공하여 PSEE 사용자들을 가이드 한다.
   - What to do next : 진행중인 프로세스의 현재 상태를 고려하여 사용자에게 다음 활동은 무엇이라는 것을 알려준다.
   - How to do "something" : 목표를 달성하기 위해 어떻게 해야 하는지의 방향을 제시한다.
   - How does "something" work : 어떻게 동작하는지를 알려준다.

[2] PCTE는 ECMA(European Computer Manufactures Association의 소프트웨어 툴의 프레임워크로서 Entity Relationship Object Management System을 근간으로 하고 있다.

- Analyze the impact of an action : 어떤 활동이 모델을 변경하였을 때의 결과를 평가 분석한다.
- 현재 달성한 것은 무엇이고 현상태가 어떻게 도달했는지를 보고한다.
- 컨트롤 기능이 사용자의 initiative를 제한할 때 어떤 일이 일어나고 있는지를 설명한다.
- 관찰을 통해, 시스템에 feedback을 제공하고 사용자가 시스템을 어떻게 사용할 수 있는지를 도와 준다.

2.5.2 ALF를 이용한 프로세스 정의와 실행

2.5.2.1 ALF의 프로세스 모델

ALF의 프로세스 모델링에 대한 관점은 다음에 정의된 6가지 주요 고려사항들로 정의될 수 있다.

1) A Configurable Open Framework 제공

프로세스가 프로젝트마다 또는 사용자마다 맞춰지기 위해서는 SEE가 기계적인 특성보다는 좀 더 사용자들의 활동과 목적에 따라 구성할 수 있어야 한다. 기존의 SEE들은 플랫폼 레벨에서 데이터의 통제, 프리젠테이션, 구조화 등의 통합 매커니즘을 제공했으나 프로세스와 프로세스의 개선을 측정하고 프로세스의 발전을 예측하고, 프로세스의 변경을 지원하고 일관성이 없고 불완전한 것을 관리하기 위한 요구사항을 만족하기 위해서는 추가적인 layer가 필요하다. 그래서 ALF는 PCTE를 기반으로 그 상위 레벨에 정의가 되었다.

2) A Multiparadigm Approach를 채택

소프트웨어 프로세스의 entity는 다음과 같이 다양하다: actions, activities, agendas, agents, configurations, deliverables, events, messages, methods, obligations, permissions, pre-post conditions, rules, triggers, types, versions, views 등. 그러므로 소프트웨어 프로세스에 대한 multiparadigm description이 필요하다.

프로세스 모델링은 시스템의 외부적인 behavior에 대한 specification과 밀접한 관계가 있는데(VDM, Merise, SSADM, Statecharts, SREM, Petri Nets), 이것들은 다음과 같은 공통된 특징이 있다.

- operations을 사용한다 -> operator model 제안
- operations은 object와 관계를 갖기도 하고 갖지 않기도 한다 -> object model 제안
- action은 events나 conditions의 통제하에 발생한다 -> model of rule 제안

- operation의 실행을 제한하는 방법을 사용한다 -> model for ordering 제안
- 프로세스의 특정한 상태를 나타내기 위해 logical constraints를 사용한다 -> notion of characteristics를 제안

3) A Structured Approach

ALF가 customizable SEE를 제공하기 위해서는 process의 definition model, enactment model 그리고 그들간의 관계를 묘사해주어야 한다.

- Process Definition Model은 사람이나 컴퓨터화된 agent의 도움으로 자원을 사용하고 어떤 규칙들을 따름으로서 product을 산출하는 개념에 의존한다. MASP(Model for Assisted Software Process)는 복잡한 소프트웨어 프로세스를 묘사할 수 있는 모델이며 이러한 모델을 instantiation하면 프로젝트나 팀 또는 특정 단체에 적합한 프로세스 모델이 생성된다. 또한 이러한 모델은 점진적으로 정의되기 때문에 model fragment의 composition이 중요하다.
- Process Enactment Model은 근본적으로 instances와 activities, 그리고 agent의 initiative로 이루어 진다. Agent에 의해서 실행 되어 지는 activities들의 set을 role이라고 한다. Enactment Engine은 여러 명의 사용자들과 상호작용하면서 그들의 작업을 도와준다. 사용자들의 작업은 actor가 initiatives를 취하고, 툴과 object를 적용하고, 규칙을 따르고, 목적을 달성하기 위한 rule을 따름으로서 수행될 수 있다.
- MASP는 5가지 tuple로 구성되어 있다(Om, OPm, Rm, ORm, C) : Om은 object model이고 프로세스 fragment에서 사용되는 모든 데이터를 나타내며 typed ERA(Entity Relationship Attribute) data model이다. OPm은 profile이 있고 pre/post condition이 있는 툴의 abstraction의 operator types이다. Rm은 rule들의 집합으로 사전에 정의된 events들이 어떻게 반응하는지를 보여준다. ORm은 path expression으로 표현되는 ordering constraints의 집합으로 operator의 순서를 표현해 준다. C는 characteristics로 invariant로 사용되는 표현이다.

MASP는 프로젝트 내에서 한명의 사용자의 activity를 모델링 해주며 이 모델이 실행되는 instance를 IMASP라고 한다. 이것을 여러 명의 사용자들로 확장하기 위해서는 model fragment들의 계층적 구조가 필수적으로 필요하다.

4) From MASPs to Enacting Model-Driven Processes

Static한 MASP 모델에서 실제로 프로세스를 실행(enacting)하기 위해서는 여러가지 단계와 개념들이 필요하다. 첫번째 단계는 instantiation으로서 MASP를 Associated Work Context로 변환하는 것이다. ALF에서는 workspace와 toolset을 만들어 IMASP를 만드는 단계이다. 두번째 단계는 work session을 생성하는 것이다. Work session은 사용자와 work context(IMASP)를 연결한 것으로 MASP-driven PSEE와

사용자와의 interface가 이 단계에서 확립된다.

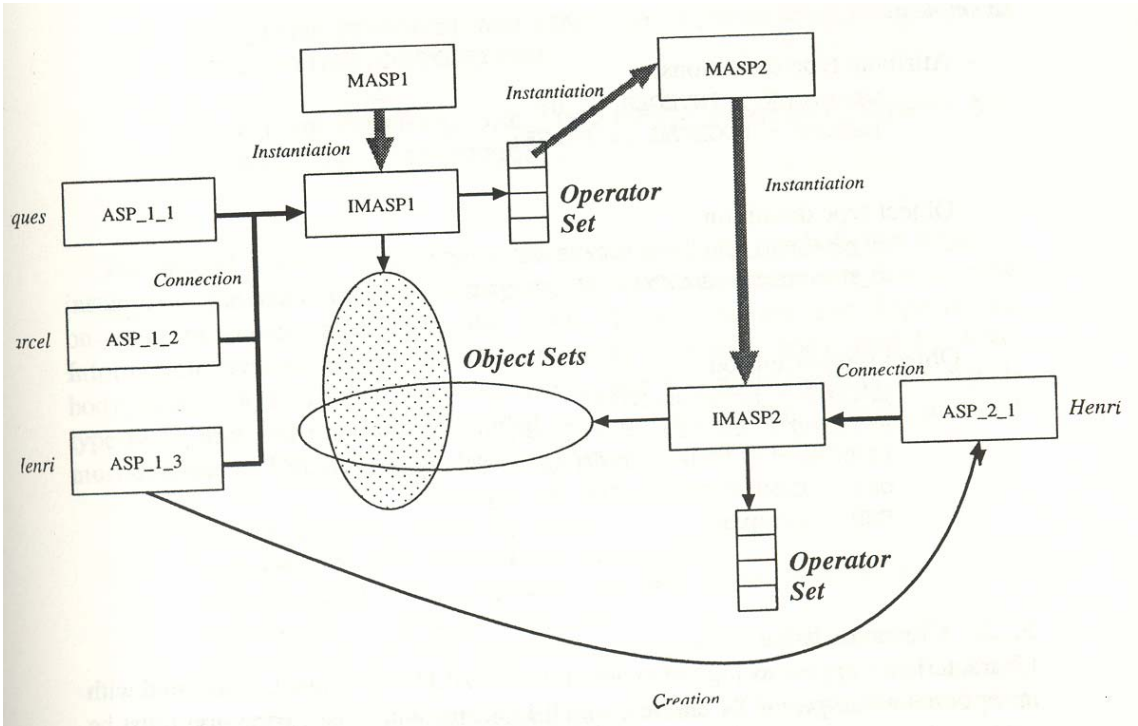
5) Team Working and Roles Cooperation 제공

Agent와 role들간의 협조를 위해서는 Shared Instance를 제공하여야 한다.

6) Process Model Evolution

프로젝트의 기간이 긴 경우 프로세스 모델이 변화하고 진화하기 때문에 프로세스 모델이 실행 중에 동적으로 프로세스 모델을 변화할 수 있어야 한다.

그림 2.5.1은 세명의 사용자가 MASP-driven 환경을 사용하는 모습을 보여주고 있는데 세명은 MASP1으로부터 instantiation 된 IMASP1 working context에 공통적으로 연결되어 있고, 각각 ASP_1_1에서(ASP: Assisted Software Process) ASP_1_3까지의 서로 다른 work session을 갖고 있다. IMASP1과 IMASP2는 그들의 work context의 일부를 공유하고 있음을 보여주고 있다.



[그림 2.5.1 : MASP-driven Environments with three users]

2.5.2.2 ALF의 프로세스 정의 언어(PDL: Process Description Language)

ALF는 소프트웨어 프로세스의 모델링을 위해서 MASP(Model for Assisted Software Process)를 사용하며, MASP는 5가지 tuple로 구성되어 있다는 것을 2.5.2.1에서 살펴보았다. MASP의 기본 구성 컴포넌트들은 relationship의 집합과 construction

mechanism으로 서로 연결되어 있다. Object-type은 aggregation mechanism으로 묘사되고, attribute-type들은 object의 특성을 묘사하기 위해 그룹화 된다. 또한 object-type들 간에는 다양한 종류의 관계(type importation, type extension, subtyping)가 표현 가능하다. Operator type과 object-type 들간의 관계는 precondition, postcondition과 같은 typed formulae를 사용하여 표현된다. 그리고 ALF는 sequence나 iteration과 같은 classical한 control structure를 제안하고 있다. 또한 ALF는 MASP들간의 관계를 묘사할 수 있기 때문에 소프트웨어 프로세서를 hierarchical한 MASP로 표현할 수 있다.

MASP/DL(Description Language)의 표현능력을 살펴보면 다음과 같다.

1) Object-Model Description Language

Object-model description language는 object types의 구조와 그들간의 관계를 표현할 수 있다.

예)

- Attribute type definitions

```
version# : INTEGER := 0;
tested : BOOLEAN := FALSE;
```

- Object type definition

```
c_program_dir : SUBTYPE OF sys_dir;
c_program : SUBTYPE PF program;
```

- Object type extension

```
EXTEND c_program WITH
ATTRIBUTE tested, version#;
LINK src : COMPOSITION LINK TO c_object;
END c_program
```

2) Characteristics

Characteristics는 logical expression(EVALUATE clause)으로 나타나고, optional한 event 파트(ON clause)와 연결되어 있다.

예)

```
ON UPDATE OBJECT body_compiled OF body_unit
EVALUATE body_unit(_b)
AND body_compiled(_b, TRUE)
```

3) Rule Model Description Language

Rule model description language는 production rule에 영향을 받았다. THEN 절 이

후는 rule이 fireable한 경우에 실행되어야 하는 action을 명시하고 있고, IF절에서는 언제 rule이 평가되고(ON 절) 어떤 조건에서 action이 실행되는지(EVALUATE 절)를 묘사한다.

예)

```
IF ON EXIT OPERATOR compile(_x)
EVALUATE successful_compilation(_x)
EVALUATED TRUE
THEN link(_x, /lib/maths.l)
```

4) Operator Model Description Language

Operator model description language는 Abstract Data Type에 영향을 받아 각 operator는 signature(type, parameters, passing mode 등)와, logical formulae(pre, postconditio 등)를 갖고 있다.

예)

```
audit   : (IN _p:project; OUT _r:report)
PRECOND : audit_authorized(_p) = TRUE
POSTCOND : audit(_p_=TRUE AND proj_to_rep(_p, _r, NO_KEY)
KIND : INTERACTIVE
```

5) Ordering Model

Ordering model은 annotated path expression으로 묘사가 되고 다음과 같은 것을 사용한다.

- Sequences : edit(x); compile(x)는 x object에 대한 edit을 먼저 한 다음에 compile을 수행
- Simultaneity : print(x)는 주어진 순간에 object x에 대한 여러 개의 print에 대한 operator invocation이 가능
- Concurrency : read(x) ||print(x)는 동시에 object x에 대한 read, print가 가능
- Grouping : (lex(x); parse(x))는 object x에 대한 lexical analysis 와 parsing을 복합적인 action으로 수행
- Alternative : print(x) | prettyprint(x)는 print 나 prettyprint가 선택적으로 수행
- Iteration : test(x)(1 TO 5)는 test가 연속적으로 5번 수행

예)

```
ORDERING_MODEL_IS
Myordering: FOR ALL _cr IN change_request DO
Create_change(_p, _cr); change_control(_cr) || estimate_cost(_cr) ||
```

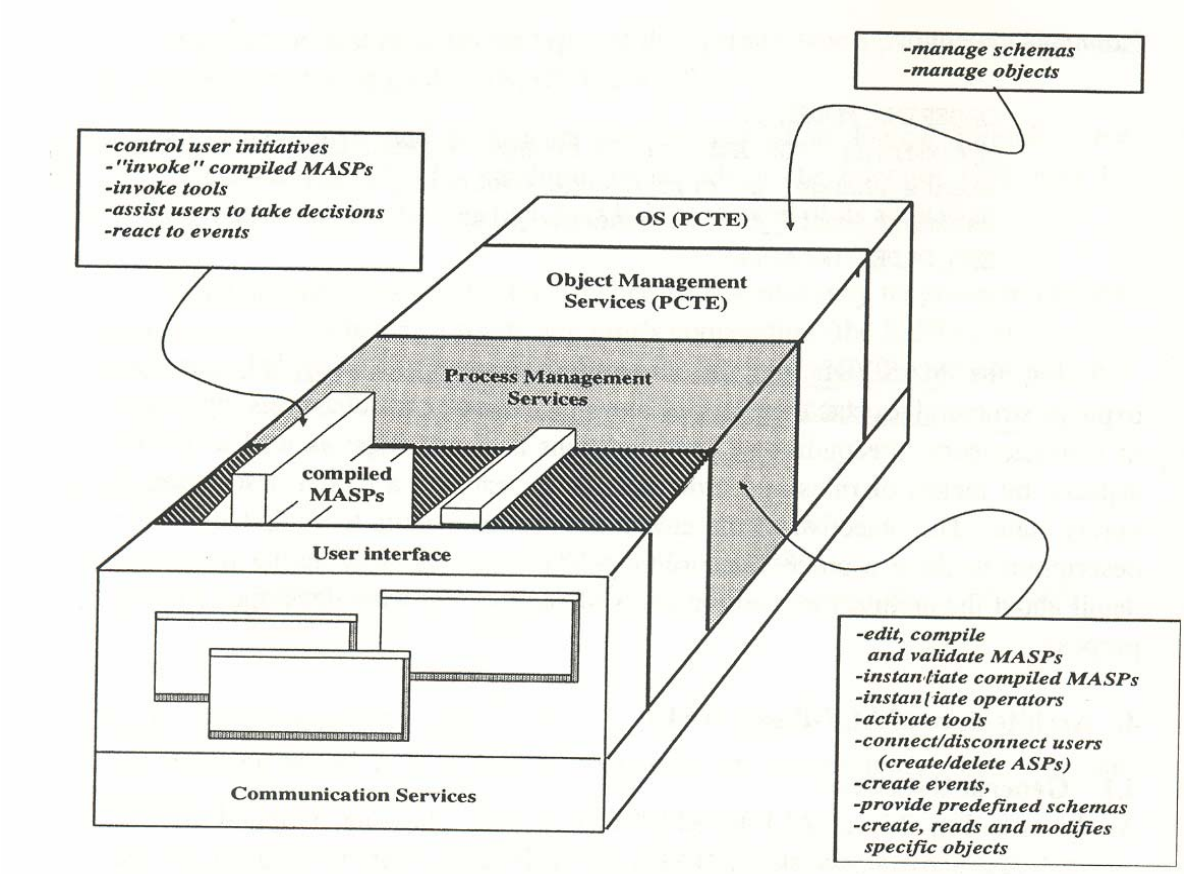
Generate_history(_cr).

END_ORDERING_MODEL

2.5.3 도구

2.5.3.1 아키텍쳐

그림 5-2는 ALF의 IPSE(Integrated Project Support Environment) 아키텍쳐를 보여주고 있는데 Process Management Services, Object Management Services, 그리고 OS는 고정된 부분이고, compiled MASP는 다른 ALF와 구분이 되는 변화가 가능한 부분이다.



[그림 2.5.2 : ALF-based IPSEs Architecture]

Compiled MASP는 MASP의 명세에 따라 사용자 또는 시스템의 initiative에 반응하고 통제, 가이드, 설명, 그리고 보고를 할 수 있는 코드들이다. Initiative에 대한 반응으로 active compiled MASP는 Process Management Service에게 compiled MASP를 instantiation 시키고 툴을 실행할 것을 요구한다. 각 layer에 대해 좀 더 자세히 살펴보면 다음과 같다.

1) Underlying Framework: PCTE

PCTE 인터페이스는 다음과 같은 세가지 주요 매커니즘을 제공한다.

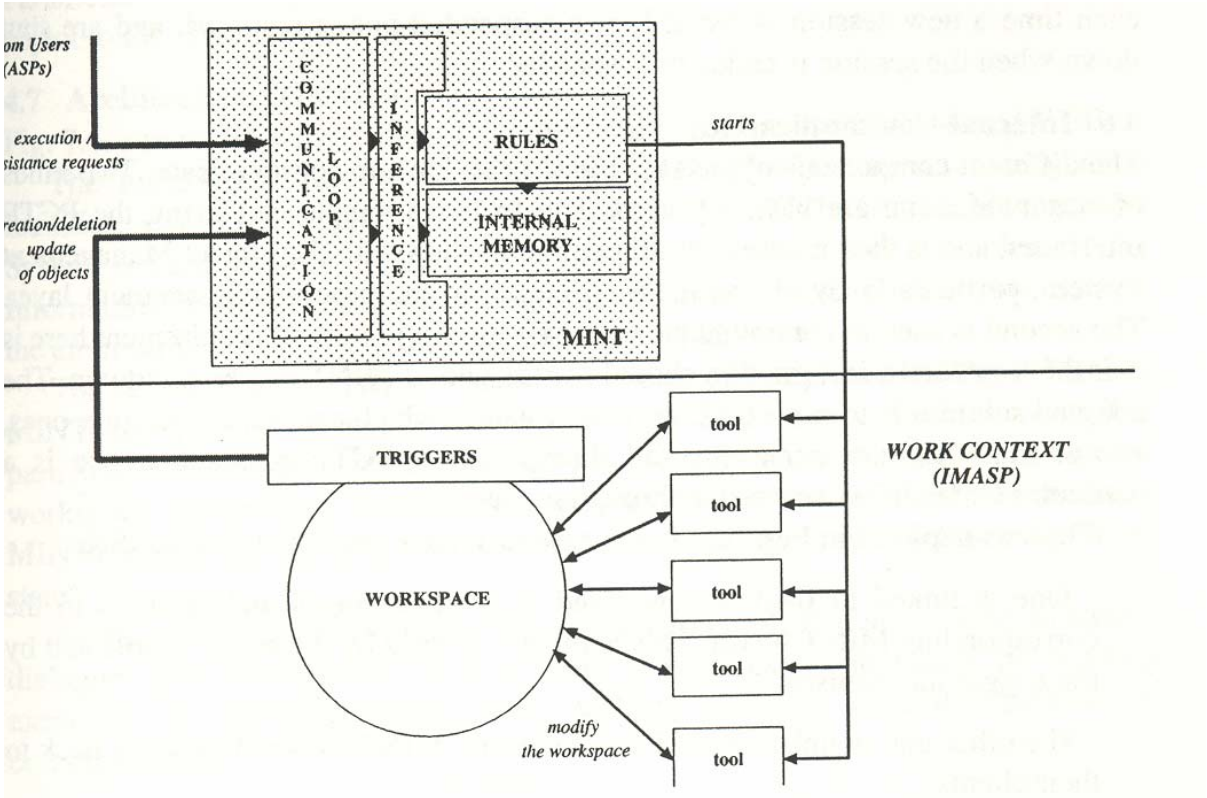
- Object Management System(OMS) mechanism
- Execution and communication mechanisms
- Distribution mechanism

2) Process Management Layer

process management layer는 프로세스 모델을 실행하기 위한 기본적인 operations을 제공하고 그 결과를 ISK라고 불리는 곳에 저장하는 역할을 한다.

3) Compiled MASPs Architecture

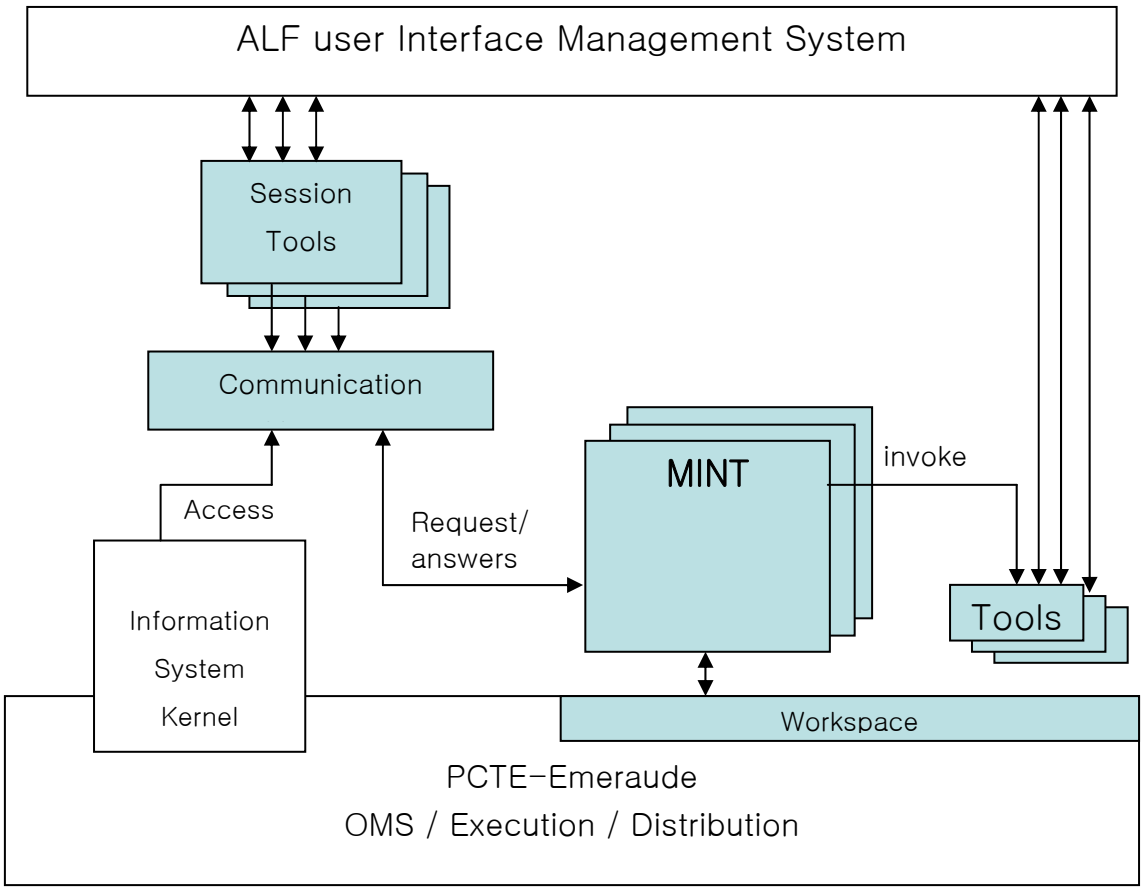
그림 5-3은 MINT(MASP Interpreter)의 아키텍쳐를 보여주고 있는데 MINT는 PCTE의 상위 layer에서 compiled MASP를 실행하는 역할을 수행하며, MASP compiler는 MASP 언어로 기술된 모델을 MINT에 적합한 schema와 input으로 변환한다.



[그림 2.5.3 : MINT Architecture]

MINT는 rule-based system으로서 MASP/DL 프로그램이 precondition을 체크하고, constraints를 ordering하는 등을 할 수 있는 rule들의 집합으로 변환된 것이고 이에 따라 MINT는 사용자의 initiative, assistance, reaction to events, 그리고 툴 invocation을 통제한다. 또한 MINT는 그림 5-3에서 보는 바와 같이 서로 다른 ASP(Assisted Software Process)들로부터의 요청을 받아 그에 대한 답을 돌려주는 서버의 역할을 담당한다.

ALF를 기반으로 한 IPSE의 아키텍쳐를 정리해 보면 그림 2.5.4와 같은데, object 관리 기능, 실행 보조, 그리고 분배 등의 기능을 제공하는 기반의 프레임워크은 PCTE가 제공한다. PCTE의 OMS(Object Management System)는 프로세스에 의해 사용되는 모든 object들을 저장하고 information system을 구현하는데 사용된다. 이 information system은 프로세스의 상태를 OMS object로 표현하여 프로세스의 영속성을 보장해 준다.



[그림 2.5.4 : ALF-based IPSE Architecture]

ALF 아키텍쳐의 핵심은 MINT인데 이것은 MASP를 compile한 것을 interpretation 하는것으로 특정 work context에서 동작하며 정의된 workspace의 object를 access하며 필요한 툴의 operation을 invoke한다. 사용자들은 시스템과의 dialogue를 제공하는 session tool을 통해서 IPSE와 작업한다. Session tool은 사용자의 명령을 MINT에 요청하고 결과를 보여주는 기능을 한다.

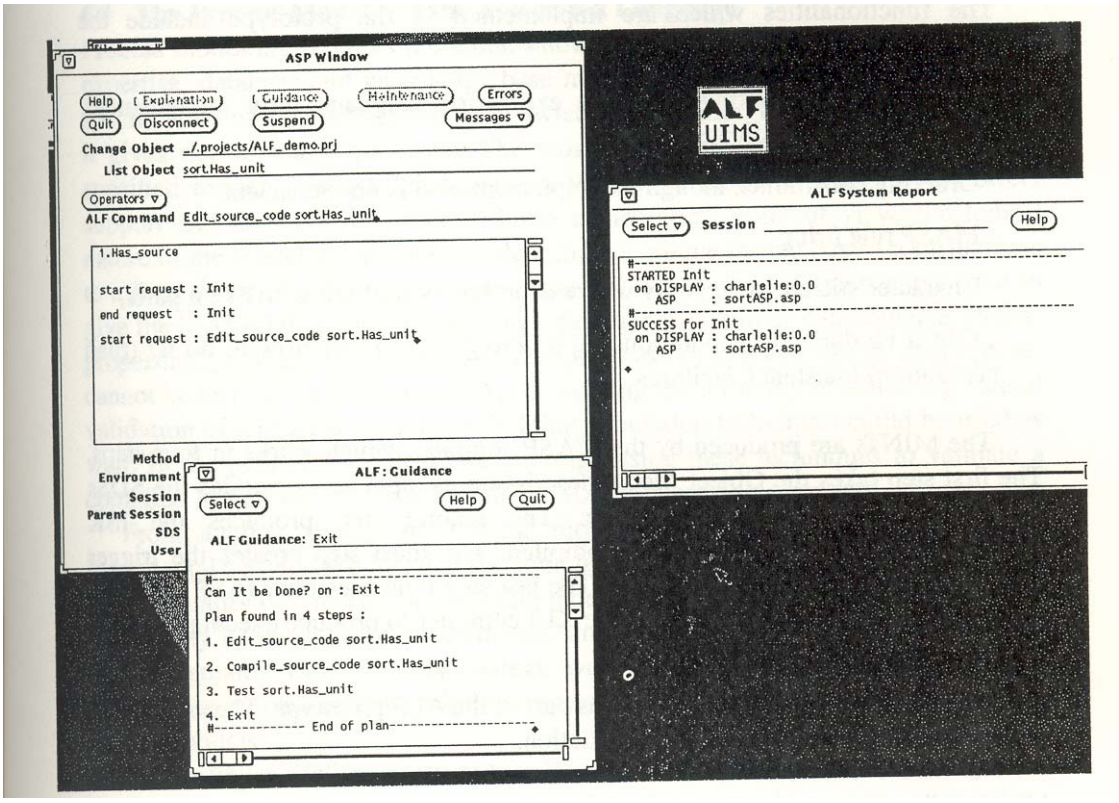
2.5.3.2 사용자의 작업환경(working context)

ALF는 그림 2.5.5와 같은 사용자 인터페이스를 제공하는데, 이 그림에서는 각 사용자들이 사용하는 주요한 3가지 session(ASP) tool을 보여주고 있다.

ASP window는 ALF command line에서 사용자가 명령을 내릴 수 있도록 해주며, session을 관리할 수 있는(quit, disconnect, suspend 등) 기능, object base navigation과 scanning(change object, list object) 기능, 그리고 session간 수행한 명령에 대한 간단한 리포트 기능 등을 제공한다.

System Report는 session이 연결된 동안 발생한 모든 events들에 대한 보고를 할 수 있는 툴이다.

Guidance는 보조기능을 제공하는 툴로서 시스템으로부터 계획을 얻는 등의 기능을 제공한다.



[그림 2.5.5 : ALF의 작업환경]

2.6 ADELE-TEMPO : An Environment to Support Process Modeling and Enactment

2.6.1 개발 배경 및 특징

대규모의 소프트웨어 시스템을 개발하고 유지할 경우 규모가 일으키는 여러 가지 관리 문제점에 직면하게 된다. 이에 프로세스 중심의 소프트웨어 공학 환경 (The Process-Centered Software Engineering Environment) 즉, PSEE가 등장했다. PSEE 분야의 연구는 다양하고 매우 많은 부분을 다루고 있으나, 여전히 몇 가지 이슈들은 어려움으로 남아 있다. Adele는 소프트웨어 프로세스가 명시적으로 분명하게 지원 가능한 커널 이라는 점에서 의의가 있으며, 소프트웨어 공학 데이터 베이스 개발에 의한 데이터와 관리 통합에 중점을 두고 있다. Adele은 객체 지향 언어와 데이터 베이스 모델링 등에 영향을 받았다. 또한, PSEE는 반드시 객체를 가지고 만든 소프트웨어를 관리하며, 활동(Activity)과 프로세스 관리에 의해 전체 프로세스 관리를 지원한다는 전제 아래에 있다. 특징으로 꼽을 수 있는 것은 데이터 베이스와 객체라고 할 수 있겠다.

2.6.2 ADELE-TEMPO의 프로세스에 대한 관점

ADELE은 객체 지향 기반의 데이터 베이스를 가지고 있다. 이 데이터 베이스에 소프트웨어 개발에 필요한 자원과, 프로세스가 모두 저장되고 관리되고 있으며, 이 데이터 베이스에서 필요한 소프트웨어에 따라 혹은 다른 버전의 소프트웨어에 따라 관련 활동(Activity)들을 관리함으로써 형상 관리를 하고 있다.
ADELE에서는 프로세스를 활동(Activity) 중심으로 모델링하고 있으며, 각 활동은 Event-Condition-Action (ECA) 이라는 규칙에 따라서 일련의 이벤트들로 표현된다. 이렇게 모델링 된 프로세스는 TEMPO라는 formalism을 통해서 실행 가능하게 되는데, TEMPO로 실행되어진 각각의 프로세스 인스턴스들은 소프트웨어를 생산하는 것이 목적인 작업 환경(Work Environment) 아래에서 활동들을 수행하게 한다. ADELE의 개념과 기술은 객체 지향 언어와 데이터 베이스 모델링, 그리고 트리거 메카니즘에 영향 받아 객체 지향의 개념을 전체 행위로 확장 시켜 사용했으며, ECA 규칙이 temporal trigger를 표현할 수 있게 확장 시켰다.

ADELE-TEMPO의 PSEE architecture는 아래와 같이 네 개의 컴포넌트로 이루어져 있다.

- Adele Object Manager (ADL-OM)

데이터 베이스 기술에 기반을 둔 발전된 객체 관리 시스템으로써 객체와 합성된 객체들의 속성 (attribute)과 객체간 관계 (relationship)가 명시적으로 설명되어 있다. 즉, 자원을 관리하는 것이다.

- Adele Activity Manager (ADL-AM)

각 활동 (activity)이 이벤트 조건 규칙 (Event-Condition-Action: ECA)에 따라 이벤트의 발생으로 모델링된다.

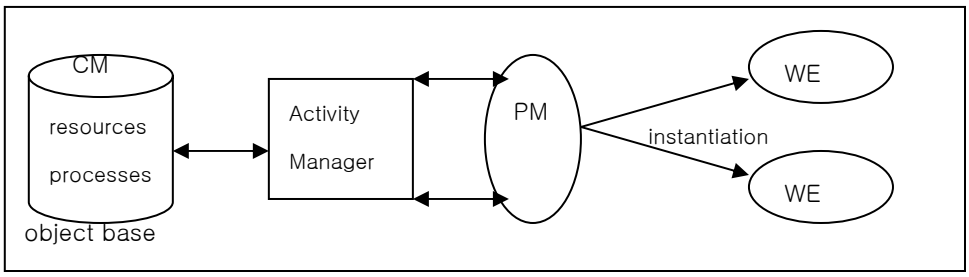
- Adele Configuration Manager (ADL-CM)

각 활동 (activity)을 소프트웨어 제품의 진화와 연결 지어 다루고 있으며, 각기 다른 버전 중에서 진화에 필요한 컴포넌트와 행위 (action)을 선택하게 한다.

- Adele Process Formalism: Tempo (ADL-TEMPO)

이것은 학문적인 연구 과제로써, 실행 가능한 소프트웨어 프로세스 형식 (formalism)을 제공한다. 이를 통해 구체적인 소프트웨어 프로세스 모델이 기술되고 실행된다. 또한 하나의 프로젝트 안에서 팀간, 혹은 팀원 간의 의사소통과 동시성 (synchronization) 을 관리할 수 있으며, 다른 소프트웨어 프로세스와 다른 작업 환경에서 사용되는 복잡한 객체간의 일관성을 조절할 수 있다.

아래 그림 2.6.1은 Adele-Tempo 의 전체 구조를 보여 주고 있다.



[그림 2.6.1 : Adele-Tempo의 아키텍처]

2.6.3 ADELE-TEMPO의 프로세스 정의와 실행

2.6.3.1 ADL-OM

데이터 베이스는 소프트웨어 제작에 필요한 것들과 프로세스에 대한 정보를 통합 관리할 수 있게 해 주는 중요한 부분이다. Adele 커널에서는 그 동안의 데이터 베이스에서 충분히 지원해 주지 못했던 것들을 포함하고 있다. Adele 커널에서는 데이터 베이스가 다음의 네 가지를 모두 관리하는 것이 가능하다.

- 많은 타입을 포함하는 커다란 객체
- 여러 버전의 소프트웨어에서의 다양한 서로 다른 객체로 이루어진 구조화 되고 합성된 객체
- 복잡하면서도 일관성을 보장하는 제약 조건
- 객체에 적용 가능하거나 다른 객체를 생산해 내는 도구 (예: 컴파일러)

Adele Data Model에서 정보는 객체들과 그들 사이의 관계로 나타내어 진다. 하나의 객체는 언제나 어떠한 객체 타입 (Object Type)의 인스턴스이고, 관계 타입 (Relation Type)은 두 객체 사이의 연결을 나타내며, 출처 객체 이름, 관계 타입 이름, 도착 객체 이름으로 (O|R|D) 이루어져 있다. 각 객체의 속성 (attribute)은 아래 그림 2.6.2과 같이 정의 된다.

```
DEFATTRIBUTE
    suffix: (p, c, h, y, l) := c ;
    system : set_of (MSDOS, Unix*, VMS);
    machine COMP := "uname –n" ; --Value is the result of the evaluation
```

[그림 2.6.2 : 객체 속성의 정의]

예를 들어 X가 Y에 의존한다는 관계를 Adele Data Model로 표현하면, (*X program depends on Y interface*) 아래 정의 그림 2.6.3과 같다. *dep*라는 관계는 프로그램과 인터페이스 사이에 혹은 인터페이스와 인터페이스 사이에서 정의되며, N:N은 객체 사이의 관계가 양방향으로 이루어짐을 나타낸다. 또한 이 관계가 DAG (direct acyclic graph)여야 함을 보여주고 있다.

```
RELTYPE dep IS... ;
 DOMAIN
    type = prog -> type = interface OR
    type = interface -> type = interface;
 CARD N:N; DAG ;
 DEFATTRIBUTE ... ;
 Triggers, Methods ... ;
END dep;
```

[그림 2.6.3 : 관계의 정의]

2.6.4.2 ADL-AM

소프트웨어 개발과 유지 보수는 계속적인 프로세스 과정을 필요로 하며 서로 다른 작업이 필요하다. 관리되는 활동 (activity)들은 복잡하고 다양한데, 예를 들면, 디자인, 문서화, 제약 조건 검사, 정책 조절 등이다. 이러한 여러 가지 활동 중 어떤 활동은 반복적이고 자동화에 적합하기도 하며, 그렇지 않은 것도 있다. Adele 에서는 반복적이고 자동화에 적합한 활동을 관리할 수 있는 활동 관리자 (activity manager)를 제안했는데, 이것은 trigger mechanism에 기반을 두고 있다.

이벤트는 데이터 베이스에서 어떤 method가 실행될 때 발생한다. 이벤트의 수를 줄이기 위해서 그리고 보다 쉬운 사용자의 프로그래밍을 위해서 Adele의 이벤트는 이벤트와 조건의 혼합으로 이루어져 있다. 아래 그림 2.6.4은 *delete_sensible*이라는 이벤트를 나타내고 있다. 이 이벤트는 공개된 컴포넌트이거나 과거의 상태가 validated인 컴포넌트를 지우려는 시도가 있을 때면 언제나 일어난다. 만약, 여러 개의 이벤트가 동시에 일어날 때는, 우선순위에 따라 실행하게 된다.

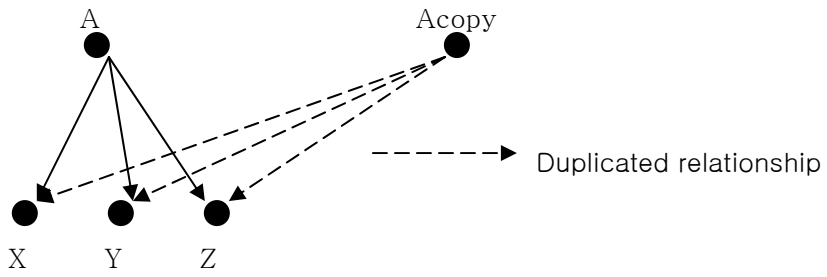
```
condition
EVENT delete_sensible = (!cmd=remove and (!object₩comp/state=released or
!object@(status=validated)); PRIORITY 5;
```

[그림 2.6.4 : delete_sensible 이벤트]

하나의 행위 (action)은 Adele 언어에서 하나의 프로그램이다. 이 언어는 간단한 명령형 언어이며, 데이터 베이스 정보의 접근과 조종에 용이하도록 구성되어 있다.

어떤 하나의 method가 실행되기 전에는 pre-trigger가 실행되고, 실행된 후에는 post-trigger 실행된다. 이러한 pre-trigger, method, post-trigger들의 집합은 데이터 베이스에서의 트랜잭션(transaction)을 구성하고 있다.

아래 그림 2.6.5는 합성 관계에서의 정의를 보여주고 있다. 원본인 A가 지워질 때는 A의 도착지인 X, Y, Z도 역시 지워져야 하며, 도착지 X, Y, Z는 스스로 지워질 수 없다. A의 복사본이 생성될 경우의 다른 컴포넌트와의 관계 또한 그림2.6.5에서 알 수 있다.




```
TYPERELATION composition;
    1 ON ORIGIN delete DO {delete !D} ;
    2 ON DEST delete DO {print "you must delete first its container !O"; ABORT};
    3 ON ORIGIN METHOD duplicate –d %new;
        {copy !O –d %new };
    4 ON ORIGIN copy DO
        {makerel %new –r %realtype –d !D} ;
```

[그림 2.6.5 : 합성 관계 정의]

아래에서 보여줄 예제는 Adele의 활동 관리자가 어떻게 개발환경의 활동을 기술하고 자동화하는지 보여 줄 것이다. 이것은 고전적인 철학자 문제이다. 철학자는 스파게티를 먹기 위해 두 개의 포크가 필요하지만 하나의 포크는 옆에 앉은 사람과 같이 써야 한다. 즉, 철학자는 먹거나 먹기 위해서 기다려야 한다.

```
OBJECT Philo is rset;
     DEFATTRIBUTE
1          STATE : (eat, think, hungry) := think ;
2     METHOD eat DO newstate (self, eat) ;
3     METHOD think DO newsate (self, think) ;
4     ERROR ON eat DO IF STATE != eat THEN newstate (self, hungry) ;
      END Philo;

OBJECT Fork is rset;
      DEFATTRIBUTE
5     STATE : (free, occupied) := free;
6     METHOD get DO IF STATE = occupied THEN {
            Print   "The %name fork is occupied"; ABORT }
            ELSE newstate (self, occupied) ;
7     METHOD release DO newstate (self, free);
END Fork ;

RELTYPE Use;
      DOMAIN type = Philo -> type=Fork ; CARD N:N ;
      PRE ORIGIN eat DO get (!D);
8     POST ORIGIN think DO release (!D) ;
      AFTER DEST release DO
9     IF ~!S%STATE=hungry THEN eat (!S) ;
END Use;

METHOD newstate (sate) DO {mda self – a STATE state } ;
```

[그림 2.6.6 : 철학자 문제의 예]

2.6.4.3 ADL-CM

소프트웨어 형상 관리 (Software configuration management: SCM) 시스템은 저장 공간 (Repository Space: RS)과 작업 공간 (Work Space: WS)사이의 협력의 결과다. WS는 SCM의 동적인 면을 조절하며, 활동(activity)들이 수행되는 공간이다. RS는 소프트웨어 객체들이 모델링되고, 저장되고 관리되는 장소이다. Adele에서는 저장 공간에 있는 자원을 꺼내서 작업 공간에서 사용하기까지의 일관성 유지의 문제들을 작업 공간이 바로 부차적인 데이터 베이스의 역할을 하게 함으로써 해결해 주고 있다. 또 다른 SCM에서의 주요 어려움 중의 하나는 필요한 구성에 알맞은 컴포넌트들의 목록을 찾는 것이다. 이것을 위해 Adele에서는 configuration builder를 제공한다. Configuration builder에서는 상품 모델 (Product Model)에 기반을 두고 있다. 이 상품 모델은 세 가지 기본적인 객체 타입을 가지고 있는데, family, interface, realization이 그것이다. 하나의 family는 하나의 모듈에 관련된 모든 변형과 모든 객체들을 모으고, 인터페이스들의 집합을 포함한다. 하나의 family에 관련된 인터페이스들의 집합은 프로시저나 타입과 같은 특질(feature)들을 정의하고, realization들을 포함한다. 각각의 realization은 기능상으로는 모두 같지만 기능적인 몇 이외의 특징은 다른데, 각각이 필요에 따라 어느 것이 낫다고 할 수 없는 평행적인 관계에 있다.
Adele에서 하나의 형상을 만들기 위해서는 완전하고 일관적인 객체의 집합과 그에 맞는 소프트웨어 시스템의 구조가 갖추어져야 한다. 이러한 시스템 모델은 최소한 몇 가지 요구사항을 만족해야 하는데 그 중에 몇 가지 예를 들도록 하겠다.
이를 테면 어떤 기능이 두 번 제공되지는 않도록 인터페이스를 제한하고, 모든 컴포넌트는 시스템 모델에 의해 정의된 제약 조건을 만족해야 한다. 그러한 제한 조건은 그림 2.6.7과 같이 표시 될 수 있다.

```
[recovery=Yes] and [system=unix] and [messages=English]
```

[그림 2.6.7: 시스템 모델 요구 사항의 예]

실제로 형상의 인스턴스는 보다 편리한 수정을 위해 적합한 제약조건에 의해 정의된다. Adele에서는 이러한 선택의 근거로 개정 특성 (revision properties)을 사용하고 있으며 그림 2.6.8과 같이 논리 언어로 표기된다.

```
([reserved=Riad] or [author=Riad] or [state=official]) and [date>18_02_89]
```

[그림 2.6.8 : revision property 의 예]

이 configuration builder와 상품 모델 (product model)은 1986년에 발표되어 많은 사용자들에 의해 사용되었고, 사용자들의 경험에 의해 아래와 같은 단점이 있다는 것이 밝혀졌다.

- 정해진 상품 모델로 모든 소프트웨어를 쉽게 모델링 할 수 없다. 특별히 오래된 상품을 다른 구조로 디자인 할 경우 interface와 realization관계를 찾는데 어려움이 있었다.

- 버전에 관한 정의와 진화에 대한 제약이 종종 너무 강하다.
- 대략적으로 비슷한 프로세스를 이용하여 소프트웨어 형상을 위한 다른 제약이 정의될 수도 있었다.
- 정해진 상품 모델이 다른 소프트웨어 프로그램에 비해 객체들을 쉽게 지원하지 않는다.

2.6.4.4 ADL-TEMPO

Adele 시스템이 높은 단계의 소프트웨어 프로세스를 기술하고 수행시킬 수 있도록 새로운 소프트웨어 프로세스 형식인 TEMPO를 소개했다. TEMPO에서는 프로세스를 모델링 하기 위해 객체 지향과 역할 개념(role concept), 그리고 ECA 규칙에 초점을 맞추고 있다.

TEMPO에서는 프로세스 모델을 역할(role)과 연결(connection), 이렇게 두 가지에 기반 하여 정의한다. 역할은 프로세스 안에서 그 역할이 담당하는 객체들의 정적이고 행위적인 특성을 동적으로 재정의 할 수 있게 하며, 연결은 프로세스들이 어떻게 협력하는지를 표현해 준다.

TEMPO의 소프트웨어 프로세스 모델은 소프트웨어 프로세스 타입의 조합으로 정의되는데, 하나의 소프트웨어 프로세스 타입은 하나의 역할을 담당하는 객체들의 집합으로 정의된다. 각각의 프로세스 타입은 활동(activity)들의 집합을 기술하고 정의한다. 이렇게 만들어진 소프트웨어 프로세스의 인스턴스는 작업 환경 (Work Environment: WE) 에서 한 사람 혹은 여러 명의 사용자에 의해 수행된다. 실제로 프로세스가 수행될 때 프로세스 사이의 통신과 동시성(synchronization)은 이벤트를 기반으로 한 ECA 규칙에 의해 기술된다. 소프트웨어 프로세스가 수행되는 동안에 객체 위에서 관리되는 모든 작업은 이벤트를 발생시키고, 이 때 정의된 규칙이 프로세스 타입을 작업 환경 하에서 사용 가능하도록 한다. 하나의 프로세스 타입은 Adele-DB에서 하나의 표준 객체 타입으로 모델링 되기 때문에 상속의 특성을 이용해 프로세스를 재정의(redefine) 하거나 특수화(specialization) 할 수 있다. 즉, 새로운 특성, 역할, 메쏘드, 규칙 들이 오버로드(overload)되거나 수정될 수 있는 것이다. 따라서, 프로세스의 맞춤(customizing)은 프로세스 타입의 특수화로 가능하다.

1) 역할 (role)

객체가 사용되는 프로세스에 따라 역할이 달라지므로, 소프트웨어 프로세스에 사용하기 위해서는 객체의 특징과 행위를 바꿀 필요가 있다. 역할 타입을 정의함으로써 원하는 대로 객체를 사용할 수 있게 된다. 하나의 역할은 객체 인스턴스들의 집합으로서 그들의 공통적인 행위와 특성을 정의한다. 여기서 특성이란 특질(attribute)를 말하고 행위란 method와 제약을 말한다. 각각의 역할은

작업 환경에 따라 본래 객체의 method를 재정의 하거나 새로운 method를 정의 할 수 있다. 다음 그림 2.6.9은 module이라는 타입이 development라는 작업 환경에서 to_consult라는 역할과 to_change라는 역할에 따라 어떻게 맞추어지는지 보여 준다.

```
OBJECT Module ;
    ATTRIBUTE
        State = tested, untested, available ;
    METHOD
        compile ...;   -- with –C option
END Module;

PROCESS development ;
     ROLE testing = unitary_testing ;
     ROLE to_consult = module ;
     ROLE to_change = to_consult/(responsible=!username);
        ATTRIBUTE state = compiled, edited, ready;
        METHOD
              compile ...;     -- with –g option
             AFTER ON compile DO test ..
END development;

PROCESS validation ;
    ROLE component = module ;
    ATTRIBUTE
        test_suite = test1, test2;
    ...
END validation;
```

[그림 2.6.9 : 역할 예제]

2) 연결 (connection)

연결은 역할 인스턴스 사이에 있다고 가정되는 특별한 관계이다. 연결은 객체들이 어떻게 협동하는지를 정의하기 위해 의도되어 졌다. 데이터 흐름의 정의가 될 수도 있고, 항상성의 검사일 수도 있고, 마감일 조절이 될 수도 있고, 메시지 전달이나 객체의 진화 조절 등이 될 수도 있다. 이러한 연결은 대칭적이지 않다.

역할간의 협력 (role collaboration)이 어떻게 이루어지는지 보기 위해 한 가지 시나리오를 생각해 보자. 어떠한 소프트웨어를 개발하고 수정할 때 소프트웨어의 모듈 M이 모든 자기 자신의 복사본이 ready state일 때만 사용 가능한 상태인 available sate가 되며, 모든 모듈들이 available state일 때만 작업 환경이 유효하다고 본다. 이러한 정책을 TEMPO로 기술하면 아래 그림 2.6.10과 같다. CONNECT라는 절은 한쌍의 객체가 연결되어있어야 함을 나타내고 있다. 6번 줄은 주어진 프로세스에서 두 개의 implement 역할이 consult_change라는 연결에 이어져 있음을 나타낸다. 이러한 연결에 따라 활동(activity)이 작업 환경 안에서 수행될 수도 있고, 소프트웨어 프로세스 과정중에 다른 활동들을 간섭하지 않을 수도 있다. 저자들은 Adele 커널을 소프트웨어 프로세스 모델을 이해하고 실행시키는 가상 기계로 사용했다. Adele 커널은 entity relationship data model에 기반을 두고 있으며 TEMPO 모델을 위한 객체 지향 기술 언어는 트리거(trigger)와 함께 객체 지향 entity-relationship model로 매핑되었다.

```
TYPEPROCESS release ;
1 EVENT ready=(state:=ready) ;
   ROLE USER=PMmanager ;
   ROLE implement=development ;
   ROLE valid = validation ;
   ROLE component = module ; {
   ON ready DO {
2 IF implement.to_change.%name.state==ready THEN
3      implement.to_change.%name.state:=available ;
4 IF implement.to_change.state==available THEN new valid;}}

5 TYPECONNECTION consult_change IS notify, resynch ;
6      CONNECT implement WITH implement
7      WHEN to_consult.name=to_change.name ;
8      EVENT notify_when=ready;
                        resynch_when=ready;
      END;

      TYPECONNECTION change_change IS notify, merge ;
           CONNECT implement WITH implement
           WHEN to_change.name=to_change.name;
           EVENT notify_when=ready;
                  merge_when=ready;
      END;

      END release ;
```

[그림 2.6.10 : 연결 예제]

2.6.4 정리

TEMPO formalism은 소프트웨어 프로세스, 특별히 프로세스 간의 협동과 자원의 분배를 위해 디자인 되었다. 이것의 주요 특징은 아래와 같다.

- 객체 지향 모델 : 프로세스와 관련된 자원들은 operation과 제약을 제공하는 객체로 모델링 되었다. 그리고 자원의 할당은 중앙 작업 공간에 의해 버전과 형상 관리에 유의하여 조절되었다.
- 역할의 개념 : 이것은 객체 지향 언어의 위임 (delegation)과 비슷한 접근 방법으로써 각기 다른 행위와 상황에 관련된 객체들을 변화시킬 수 있는 새로운 메커니즘을 제공한다.
- 연결의 개념 : 프로세스 사이의 동시성(synchroization)과 협동을 명시적으로 표현해 준다.

TEMPO는 Adele 시스템의 가장 상위에 구현되어 있으며, TEMPO로 기술된 소프트웨어 프로세스 모델은 객체 타입과 관계, 이벤트 규칙으로 이루어진 Adele의 개념으로 해석된다. 또한 프로세스 실행을 지원하는 활동 관리(activity manager)의 사용으로 method들은 수행 순서에 따라 조건에 합당할 때만 수행되게 된다.

2.7 SPADE : An Environment for Software Process Analysis, Design, and Enactment

2.7.1 개발배경 및 특징

SPADE는 Software Process Analysis Design and Enactment를 지원하기 위한 software engineering environment를 제공할 목적으로 Poliecnico di Milano와 CEFRIEL에서 수행된 프로젝트이다. SPADE는 다음과 같은 특징들을 제공한다.

- SLANG이라고 하는 Petri-net 기반의 Process modeling language를 제공한다.
- Software artifacts와 process models등을 포함한 process data를 저장하기 위한 object-oriented database인 O_2에 기반한 interpreter를 제공한다.
- Process에서 fine grained tool integration을 달성하기 위해서 DEC FUSE라는 message based integrated toll environment를 사용한다.
- Distributed development activity들을 지원한다.
- Meta-process뿐만 아니라 software process의 specification과 enaction을 지원한다.
- SPADE는 process evolution을 효과적으로 지원하는 mechanism을 제공한다. SPADE는 process definition들과 process state들에 대한 change를 지원하는 reflective feature들을 제공함으로써 process evolution을 지원한다. 이러한 process evolution은 process가 enacted된 상태에서도 가능하다.

2.7.2 SPADE의 SLANG를 이용한 process 정의와 실행

2.7.2.1 SLANG의 구성요소

SLANG은 process modeling을 위해 Petri-net을 사용한다. 이러한 측면에서 process는 여러 개의 activity들로 구성되고 각 activity들은 Petri-net에서 하나의 net로써 표현된다. 좀 더 구체적으로 얘기하면 각 activity는 interface와 implementation을 가진 SLANG net이 된다. SLANG net은 일반적인 Petri-net의 net처럼 token, place, transition, arc를 가진다. 각 구성요소를 간단히 요약하면 아래와 같다.

- Token : process에서 다루어지는 object들을 나타낸다.
- Place : token을 저장한다.
- Transition : process에서 수행할 event들을 나타낸다.
- Arc : process에서 발생하는 events와 objects의 flow를 나타낸다.

2.7.2.2 SLANG의 process 정의와 실행

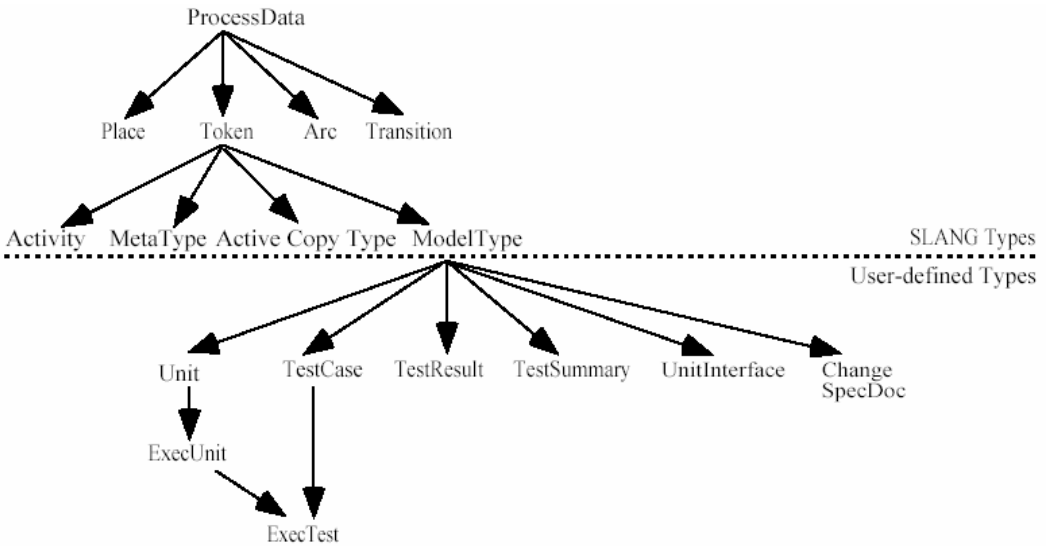
process model은 실제 software process와, process model을 개발하기 위해 사용하는 meta process를 구성하는 activity들과 task들을 포함해야 한다. 그러므로 process model을 사용하는 process machine은 software process를 자동화하기 위해서 software process를 수행될 code로써 또한 process model를 수정하기 위해서 edit될 data로써 사용할 수 있어야 한다. execution과 editing activity들의 interleaving은 process내에서 수행되고 있는 operation들을 시작하는 것을 중단하거나 restart 시키지 않고 old process model에서 new process model로 자연스럽게 넘어갈 수 있는 방법으로 관리되어야 한다. 이러한 process model evolution을 효과적으로 달성하기 위해서는 language가 reflective feature들을 제공해야 한다. SLANG은 process model의 수정과 evolution을 관리하기 위한 meta-process를 model하기 위한 reflective feature를 제공한다. 앞으로 SLANG을 이용하여 process를 기술하는 방법과 process evolution을 위해 어떠한 reflective feature들을 제공하는지에 대해 설명할 것이다.

2.7.2.2.1 SLANG의 process description

SLANG의 process specification은 다음과 같다.

SLANGSpec = (ProcessTypes, ProcessActivities)

ProcessTypes는 type hierarchy에 있는 type description들의 set이다. 각 type은 object oriented style로 기술된다. ProcessActivities는 type Activity의 instance들의 set이다. ProcessTypes의 hierarchy는 아래와 같다.



[그림 2.7.1 : ProcessTypes hierarchy]

위의 hierarchy에서 SLANG이 Petri-net에 기반하고 있기 때문에 Place, Transition, Arc등의 type이 존재한다. Token은 net에서 token을 기술하고 ModelType, Activity, MetaType의 3가지 subtype을 가진다.

1) ModelType

ModelType은 process내에서 다루어질 object들을 기술하는데 필요한 모든 type의 root이다. ModelType의 subtype들은 특정 process마다 달라지고 사용자에 의해 결정된다.

2) Activity

Activity는 places, transitions, arcs로 구성된 Petri net으로써 SLANG activity를 정의한다.

3) Metatype

Metatype은 type들이 어떻게 기술되어져 있는지를 나타낸다. Token의 각 subtype에 대해 하나의 Metatype instance가 존재한다.

Type definition에 대한 예제는 다음과 같다.

```
Class Unit inherit ModelType
type tuple   (public name : unitName, public authorName : personName,
               public language : string, public sourceCode : text)
end;
```

[그림 2.7.2 type Unit에 대한 Type definition 예]

Activity 정의는 interface part와 implementation part로 구분된다. Activity는 interface를 통해서 다른 activity와 상호작용 한다.

Interface part는 interface transitions set, interface places set, arcs set으로 구성된다.

1) interface transitions

interface transtitons는 starting events를 나타내는 transitions와 ending events를 나타내는 transitions로 나누어진다. Activity는 starting event가 발생할 때 시작해서 ending event가 발생할 때 종료된다.

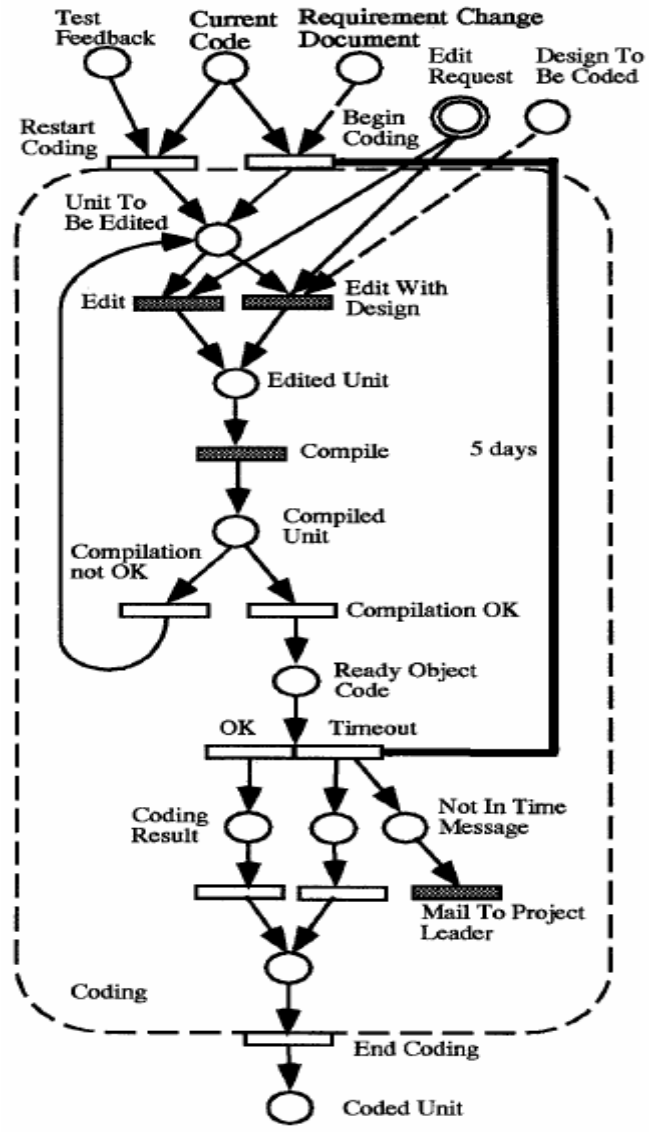
2) interface places

interface places는 input places와 output places, shared places로 분류되고, Implementation part는 어떻게 input들이 output들로 바뀌는지를 기술한다.
Activity definition에 대한 예제는 아래의 시나리오에 기반해서 보여질 것이다.

activity는 process의 coding 단계를 기술하는 activity이다. 사용자는 unit code를 검증된 design을 가지고 하든지 design없이 하든지의 2가지로 editing할 수 있다. Code가 작성된 이후에 컴파일되고 process가 컴파일 동안에 에러가 없을 때까지 반복된다. 만일 컴파일 된 unit이 5일 이내에 준비되지 않으면 project leader가 email을 통해서 통보를 받게 된다.

[그림 2.7.3 : 예제 시나리오]

activity의 interface part는 점선의 사각형 바깥에 존재하는 entities이고 implementation part는 점선의 사각형 내부에 존재하는 모든 entities이다. 위의 시나리오에서 starting events는 {Restart Coding, Begin Coding}, ending events는 {End Coding}으로 Test Feedback, Requirement Change Document는 input places로 Coded Unit은 output place로 둘 수 있다. 한편 software process는 다양한 software tool들의 activation을 포함한다. 이러한 tool invocation은 black transition으로 나타내다. Black transition은 asynchronous하게 동작한다. 위의 시나리오에서는 transition Edit, Compile, Mail To Project Leader로 둘 수 있다. 위의 시나리오에 대한 자세한 activity는 그림2.7.4처럼 나타난다.



[그림 2.7.4 : SLANG implementation of the “coding” activity]

2.7.2.2.2 SLANG의 reflective features

SLANG은 효율적인 process evolution을 위해서 다음과 같은 reflective features를 제공한다.

1) Concurrent interpretation

Activity가 다른 activity에 의해서 호출되거나 black transition이 수행될 때마다 각 activity의 active copy들이 서로 다른 process engine에 의해서 독립적으로 수행된다. 이러한 이유로 인해서 activity의 definition이 변경된 이후에 생성된 active copy들은 변경 전에 생성된 active copy에 영향을 주지 않고 독립적으로 동작할 수 있다.

2) Late-binding

SLANG은 다음과 같은 dynamic binding rule을 제공한다.

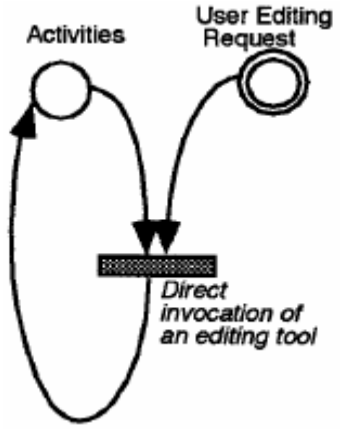
(1) 모든 definition이 run-time에 bound된다. 따라서 type checking이 완전히 dynamic이다.

(2) 같은 activity definition에 대해서 서로 다른 version의 type definition이 서로 다른 active copy들에서 동시에 사용될 수 있다. Type과 activity definition의 변화들은 그들이 사용될 때만 유용하다. 즉, enacted process에서 definition의 변화와 새로운 definition의 사용간에 latency time이 존재한다.

3) Re-entrant mechanism

모든 activity는 activity place에 저장되어 있다. Activity definition을 수정하기 위해서는 activity place에서 수정될 activity의 token을 read한 후 그 token을 activity editing toll에 넘겨준다. 이 tool은 Activity place로 다시 저장된 순간부터 유용한 새로운 version의 token을 만들고 activity place에 저장할 때 이전의 token과 바꾼다. 이후에 해당 activity에 대한 active copy들은 새로운 version의 activity definition을 이용한다.

요약하자면 process가 enacted상태일 때 process의 일부분을 수정하려고 하면 activity place에서 수정할 activity token을 read한 후 black transition을 통해 activity editing tool을 호출한 후 editing tool을 이용하여 activity definition을 수정한 후 수정된 activity를 activity place에 저장한다. 이후에 해당 activity가 호출되면 새로운 version의 active copy가 생성되고 이 때 각 process engine이 독립적으로 수행되고 dynamic type checking이 이루어지므로 이전 version의 active copy와 새로운 version의 active copy가 서로에게 영향을 주지 않고 독립적으로 수행된다.
SLANG은 위와 같은 reflective feature들을 제공함으로써 process가 enacted 상태일 때에도 dynamic evolution을 제공해 줄 수 있다. SLANG의 process evolution에 대한 대략적인 그림은 그림2.7.5와 같다.



[그림 2.7.5 : activity를 나타내는 token의 editing]

2.3.3 SPADE Architecture and Tools

2.3.3.1 Architecture

SPADE는 process model interpretation과 user interaction간에 separation of concerns principle을 적용했다. 이 principle에 따라 SPADE는 three layer architecture에 기반한다.

1) User interaction environment

User interaction environment는 environment내의 tool을 통해서 user와의 interaction을 수행하는 역할을 한다. 이 layer에는 software development activity들을 달성하기 위해서 SPADE user에 의해 사용되는 tool들이 포함되어 있다. 또한 user interaction environment에서 event를 일으키고 event들이 수행될 SLANG activity들에서 token으로 보이게끔 하는데 사용되는 filter를 포함한다.

Filter는 SLANG interpreter에 의해 발생한 service 요청들을 시작하게 하고 그 요청들을 적당한 tool들로 forward한다.

2) Process enactment environment

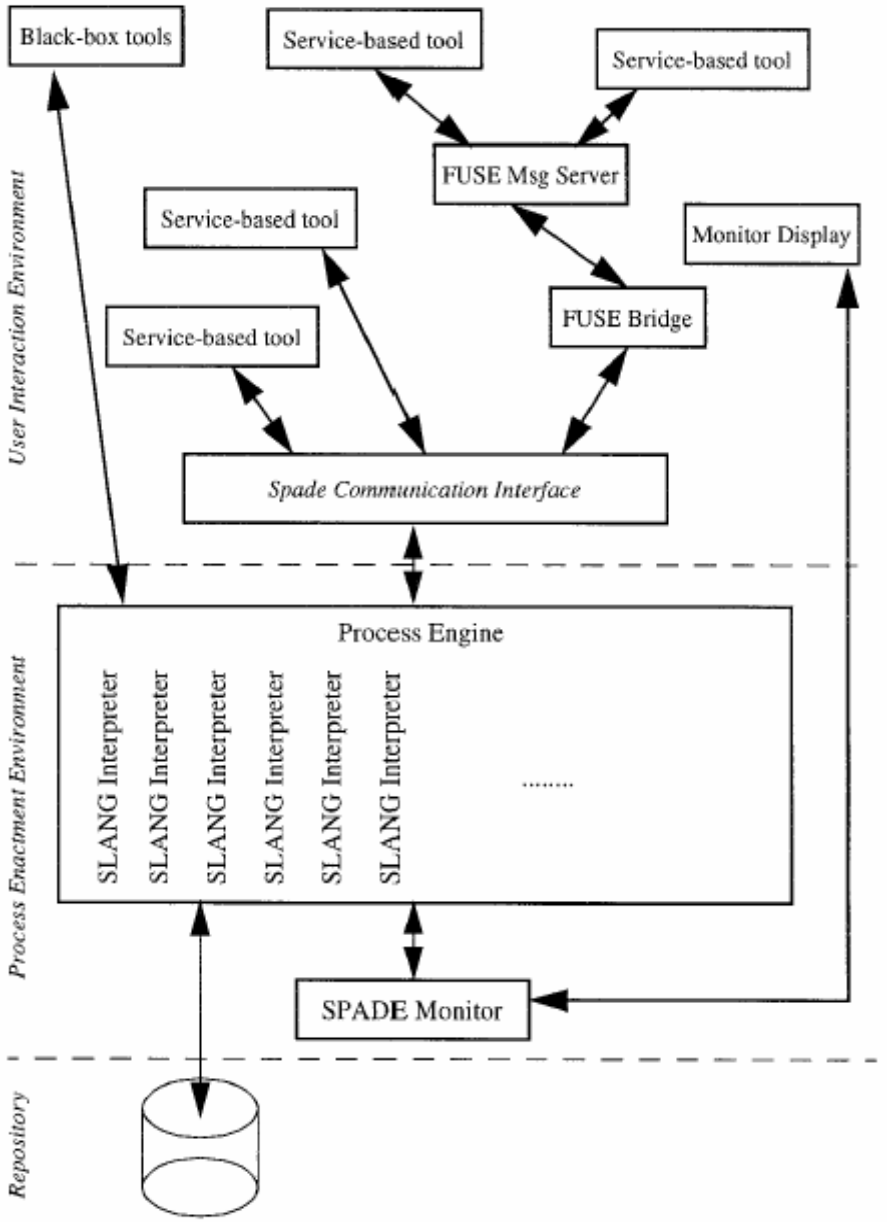
Process enactment environment는 process model을 수행하는 역할을 한다. 각 수행은 enactment동안 동적으로 생성되는 process engine들의 set에 의해 동시에 수행된다. 이 layer에는 process engine으로 불리는 Unix process들의 thread들로써 구현된 SLANG interpreter들이 존재한다.

3) Repository

Repository는 process model들과 process artifact들을 저장한다. Repository는 O2 OODBMS상에서 구현된다. O2는 client-server architecture를 가지고 있고 SPADE의 evolution feature들을 지원하는데 적합한 programmatic interface를 제공한다.

SPADE에서는 각 process engine들이 client가 된다.

SPADE의 architecture는 그림 2.7.6과 같다.



[그림 2.7.6 : SPADE architecture]

2.3.3.2 Tools

SPADE에 포함된 tool들은 크게 4가지로 분류될 수 있다.

1) Black-box tools

Black-box tool들은 process enactment environment의 입장에서 어떤 input을

받아서 output을 만들어내는 함수로 보여진다. Control integration측면에서는 environment가 이 tool들을 호출하고 종료하는 것을 관리한다. Data integration 측면에서는 이러한 tool들이 file system과 같은 각자의 repository를 사용한다. 따라서 tool들이 O_2 DB에 저장된 process data를 사용하기 위해서는 명시적으로 바뀌어지고 바뀐 내용이 file system에 반영되어야 한다. Black-box tool들은 source code의 수정없이 바로 SPADE로 integrated될 수 있다.

2) O_2-based black-box tools

이 tool들은 O_2 object들을 별다른 변형과정없이 직접 다룰 수 있다. SPADE는 ICCM이라고 하는 inter-O_2 client communication mechanism을 통해서 이러한 tool들과 data를 교환한다.

3) Service-based tools

이 tool들은 자신들이 제공하는 service들을 따로따로 호출할 수 있도록 하는 programmatic interface를 제공한다. Tool들은 다른 tool들이 제공하는 service들을 호출할 수 있다. Service-based tool들은 SPADE에 두가지 방법으로 integrated된다. 첫번째는 SCI(Spade Communication Interface)에 의해서 제공되는 SPADE Integration Protocol을 직접 사용하는 것이고 다른 하나는 FUSE나 Tooltalk과 같은 Commercial-Off-The-Shelf를 사용하는 것이다. 후자의 경우에 message들은 tool integration environment의 message broadcasting server에 의해 관리된다. 이러한 모든 message들은 특정 Bridge라고 하는 SPADE component에 의해 받아들여질 수 있다. Bridge는 message를 SPADE Integration Protocol로 바꾸고 SCI로 보낸다. SPADE는 이러한 DEC FUSE, Tooltalk, DDE, OLE2의 bridge들을 제공한다.

4) O_2 service-based tools

이 tool들은 service-based integration을 제공하는 programmatic interface와 O2 object들간의 direct exchange를 지원한다.

아래의 table은 SPADE에 integrated 되거나 SPADE가 제공하는 tool들을 위의 4가지 category에 따라 분류한 것이다.

	black-box	O_2 black-box	service-based	O_2 service-based
SPADE Shell			•	
Agenda			•	
Object Editor		•		
SLANG Editor				•
FUSE Editor			•	

FUSE Builder			•	
Emacs, cc	•			

[표 2.7.1 : Examples of SPADE Tools]

각 tool에 대해 간단히 살펴보면 우선 SPADE Shell은 process agent들이 process enactment environment를 통해서 명령을 보낼 수 있는 interface이다. Agenda는 사용자에게 할당된 일들의 목록들을 보여준다. Object Editor는 User가 SPADE Repository에 저장된 어떤 process artifact을 편집할 수 있게 하는 tool이다. SLANG Editor는 SLANG process model의 편집과 컴파일을 지원한다. FUSE Editor는 service-based text editor이다. FUSE Builder는 integrated C compiler와 make utility이다. Emacs와 cc는 Unix 환경의 editor와 C compiler이다.

한편 위의 4가지 범주의 tool외에 SPADE Monitor는 사용자에게 process model의 실행상태를 보여준다. 또 SCI는 모든 service-based tools, Bridges, SLANG Interpreter들을 수행하는 Process Engine과 연결되어 SPADE Integration Protocol을 통해서 service-based tool들과 process enactment environment간의 통신을 담당한다. 또한 User Interface Environment에서 실행중인 active copy들로 message들을 넘겨줄 수도 있다.

2.8 PEACE : Goal-Oriented Logic-Based Formalism for Process Modeling

2.8.1 개발배경 및 특징

PEACE(Process-centred Enactable and Adaptable Computer-aided Environment)는 Pierre Mendes France대학의 CRISS 연구 센터에서 진행된 연구 프로젝트의 결과로 제안되었다. 이 연구의 목적은 프로세스를 PEACE/PDL(PEACE Process Definition Language)로 정의 하여, 소프트웨어 제품을 디자인하고 개발하는 프로세스를 제공할 수 있는 자동화 환경을 구축하는 것이었다. 특히 PEACE 프로젝트에서는 소프트웨어 프로세스의 dynamicity와 evolvability를 보장해주는데 중점을 두었고, 이를 위해 소프트웨어 프로세스 모델링에 goal-oriented approach 와 logic 기반의 formalism을 채택하였다. 따라서 프로세스를 모델링 하는데, 설정된 goal을 만족시키기 위해 프로세스 모델을 dynamic하고, evolvable하게 구성할 수 있도록 하는데 중점을 두었고, 지식 기반의 정보를 통해 최적화된 프로세스 모델을 구성할 수 있도록 하는 특징을 갖고 있다.

2.8.2 PEACE를 이용한 프로세스 모델링

2.8.2.1 PEACE를 이용한 프로세스 모델링

PEACE/PDL을 이용한 프로세스 모델링은 다음과 같이 이루어진다.
PEACE Process Model 은 PMF(Process Model Fragments)들의 집합으로 이루어진다. 여기서, PMF는 관련된 goals을 포함하고 있는 프로세스의 step으로 구성되는데, 프로세스 모델을 명세할 수 있는 PMF Specification과 실제 프로세스 모델을 구현하는 PMF Implementations으로 구성된다.

1) PMF Specification

- PMF Specification은 object와 operator로 표현된다.
- Specification-identifier, I_role(Intrinsic Role), inputs, outputs, I_precondition과 I_postcondition, In_event와 Out_event로 구성되며 다음과 같은 의미를 갖는다.
  - Specification-identifier : 프로세스 스텝의 이름을 나타낸다.
  - I_role : PMF 실행중 담당하게 될 역할에 대해 나타내고 있으며, 전체적인 개발 전략에 의해 영향을 받는다.
  - Inputs/outputs : PMF에 입력되고 출력되는 object들의 이름과 타입을 나타낸다.

- I_preconditions/I_postconditions : PMF 실행전/후에 반드시 만족시켜야 하는 조건들을 나타낸다.
- In_event/Out_event : PMF에 의해 처리되거나 발생하는 event들을 나타낸다.
- PMF Specification은 다음과 같이 표현된다.

<specification>::= specification <specification-identifier>
'('<input>')' '→' '('<output>')' is
[I_role <role-expression>]
[I_precondition <condition-expression>]
[I_postcondition <condition-expression>]
[In_event <event-expression>]
[Out_event <event-expression>]
end <specification-identifier>

2) PMF Implementation

- 우선 Complex 한 PMF implementation은 contextual rule과 event handler들이 캡슐화된 operator모델로 구성된다.
- List-of-encapsulation, C_role(Contextual Role), C_preconditions과 C_postconditions, contextual_rules, event_handlers로 구성되며 다음과 같은 의미를 갖는다.
  - List-of-encapsulation : PMF_S 들의 리스트들을 캡슐화 시켜 나타낸다.
  - C_role : PMF가 실행 도중 담당하게 될 역할에 대해 기술하고 있으며, 특정한 프로젝트의 특성에 의해 영향을 받는다.
  - C_preconditions/C_postconditions : 특정한 프로젝트의 PMF 실행전/후에 반드시 만족시켜야 하는 조건들을 나타낸다.
  - Contextual_rules : PMF를 Implement할 때 필요한 논리적인 규칙들을 기술한다. 즉, 특정한 프로젝트의 특성에 따라, 어떤 조건들을 어떤 순서로 수행해야 한다는 등의 규칙에 대해 logical expression들로 표현하는 것이다.
  - Event_handlers : 특정한 event에 대해 처리할 때 필요한 action들을 나타낸다.
  - PMF Implementation은 다음과 같이 표현된다.

<complex-implementation>::=
implementation <implementation-identifier> of
<specification-identifier> is
<list-import>
[<list-of-encapsulation>]
[contextual_rules<expression>]

[event_handlers<treatment>]
end<implementation-identifier>

<list-import> ::=
<item-importation>|<item-importation> ';' <list-import>

<item-impotation> ::=
import<specification-identifier> '('<parameter-types> ')'
'→' '('<parameter-types> ')'
as<specification-identifier> '('<parameter-types> ')'
'→' '('<parameter-types> ')']

<list-of-encapsulation> ::=
<encapsulation>|<encapsulation> ';' <list-of-encapsulation>

<encapsulation> ::=
encapsulate <specification-identifier> with
C_role <role-expression>
C_precondition <condition-expression>
C_postcondition <condition-expression>
end <specification-identifier>

<treatment> ::= <on-do> ';' <treatment> | <on-do>
<on-do> ::= on <expression> do <expression>

- PEACE 프로세스 모델에서, pre/post condition, contextual rules, 그리고 events 와 logical expression들은 PEACE/PDL 이라는 logical한 language로 표현된다.
  - PEACE/PDL은 형식성과 확장성을 동시에 보장해 주는 특성을 지니고 있다.
  - Logical Language는 다음과 같은 구성요소를 갖추고 있다.
    - ✓ Constants의 집합 set C
    - ✓ Variables의 집합 set V
    - ✓ Function symbols의 집합 set F
    - ✓ Predicate symbols의 집합 set P
    - ✓ Connectives : $\Rightarrow$, $\wedge$, $\vee$, $\neg$
    - ✓ Quantifiers : $\exists$ and $\forall$

- ✓ Punctuation marks : (,)

- Language 의 구성은 다음과 같다.
  - ✓ Terms : Language의 term은 C의 element이거나, V의 element이거나 expression function이다.
  - ✓ Atomic formulars : Language의 atomic formular는 PREDICAT $_n(t_1, t_2,..,t_n)$ 으로 표현된다.
  - ✓ Well formed formulars : Language의 정의는 recursive하게 되어 있다.
- 이러한 Language를 이용하여 Precondition을 표현하면 다음과 같다.
  - ✓ Requirement의 변환 문서인 R에 따라, 모듈 M과 관련된 디자인 문서인 D를 수정하는 activity 가 있다고 가정하자.
  - ✓ 이 activity의 precondition은 "수정과 관련된 모듈 M은 requirement의 변화를 기술한 문서 R을 조회하여, 디자인 문서 D를 만들어내야 한다." 라고 했을 때,
  - ✓ ∃ M (MODULE(M) ∧ DESIGN(M, D) ∧ REQUIREMENT(M, R)) 과 같이 표현할 수 있다.

- PEACE Process Model의 예를 들면 다음과 같다.
  - Software process change의 technical한 step에 대한 예제 (CHANGE_PROCESS STEPS)
    - ✓ Requirement change document에 의해서 Design이 수정되는 상황에 대해 PMF specification과 PMF implementation을 하면 다음과 같다.
    - ✓ PMF TECHNICAL-STEPS specification

```
sds MODIFY_DESIGN;
specification MODIFY_DESIGN (TE-MD:ROLE-MODIFY_DESIGN;
    D: DESIGN; R: REQUIREMENT-CHANGE) →(D*:DESIGN) is
l_role ∃ DE (DESIGNER(DE) ∧ TEAM-DESIGN(TE-MD, DE))
l_preconditions
    ∃ M (MODULE(M) ∧ DESIGN(M, D) ∧ REQUIREMENT(M, R))
l_postconditions
    NEW-DESIGN(D, D*)
end MODIFY_DESIGN
```

    - ✓ PMF TECHNICAL-STEPS Implementation

```
Implementation TS of TECHNICALS-STEPS is
    import MODIFY_DESIGN(TE-MD: ROLE-MODIFY_DESIGN;
        D : DESIGN; R:REQUIREMENT-CHANGE) →(D*:DESIGN);
    encapsulate MODIFY_DESIGN with
        C_preconditions ¬B(¬STATUS(correct, D*))
```

C_postconditions STATUS(correct, D*)

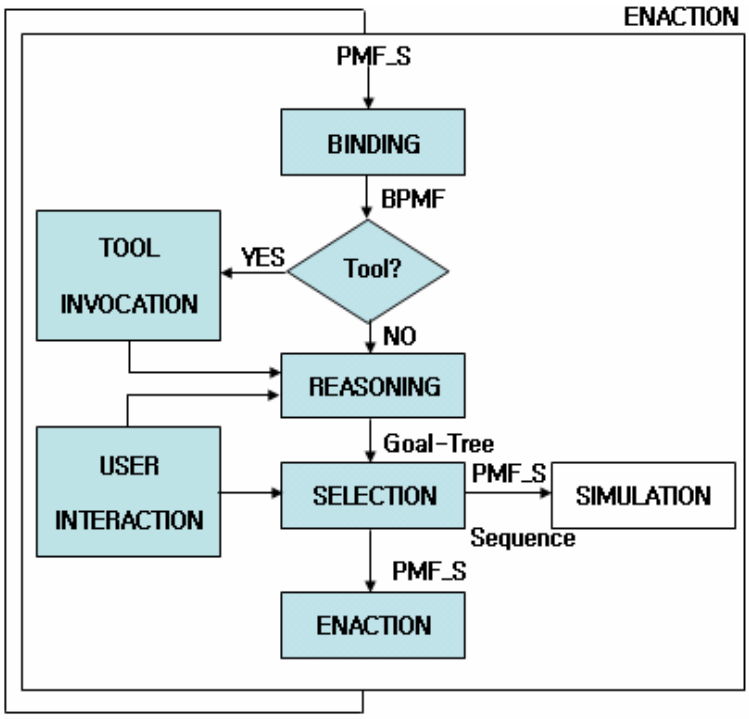
end MODIFY_DESIGN

end TS

2.8.2.2 PEACE를 이용한 프로세스 실행

PFM을 통해 생성된 Process model은 PEACE 환경하에서 Binding, Reasoning, Selection, Simulation and Enaction의 절차에 의해 실행된다.

PEACE 환경하에서 실행되는 절차를 정리하면 그림 8.1과 같다.



[그림 2.8.1 : Enaction Steps]

1) Binding

Binding은 PMF specification 인 PMF_S를 구현 가능한 PMF implementation인 PMF_I 로 구성하는 단계이다. 하나의 PMF_S는 여러가지 PMF_I로 구성할 수 있다.

2) Reasoning

구축된 프로세스 지식을 기반으로 하여 goal-tree(an and/or graph) [2] 라는 것을 만들고, 이것을 이용하여 goal을 달성하기 위해 필요한 하위 프로세스 조각들을 조합할 수 있는, 가능한 경로들을 제공해 준다.

3) Selection

프로세스 수행자가 PMF_S 의 순서를 선택하거나, PMF_S를 찾기 위한 전략을 선택하는 단계이다.

4) Simulation

사용자에게 실제로 선택한 프로세스 단계를 수행할 것인지, 다른 순서나 전략을 택할 것인지의 판단을 도와주기 위한 단계이다.

5) Enaction

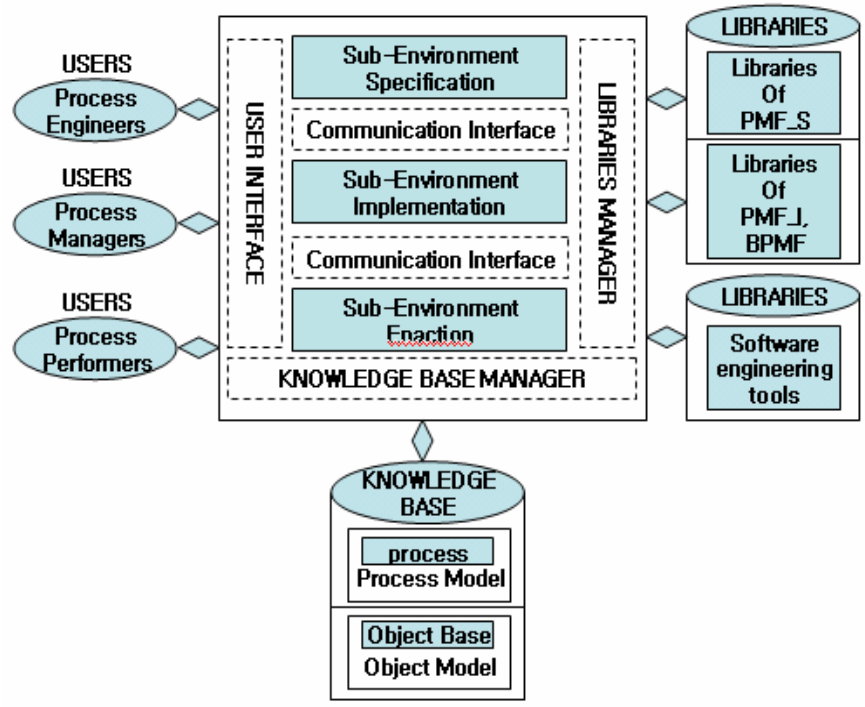
최종적으로 선택된 PMF_S를 선택된 순서에 의해 수행하는 단계이다.

2.8.3 PEACE 아키텍쳐

2.8.3.1 아키텍쳐

PEACE 환경은 Specification, Implementation과 Enaction의 세개의 하위 환경으로 구성된다. 이 세개의 하위 환경은 각기 소프트웨어 프로세스 모델링을 지원하게 되는데, 일반적인 프로세스 모델을 개발하고, 이러한 일반적인 모델을 특정한 프로젝트에 맞게 최적화 하고, 이렇게 최적화된 모델을 실행하는 방식으로 진행되게 된다.
PEACE 환경의 구조는 다음과 같이 구성된다.

- 프로세스 모델의 지식 기반이 저장되어 있는 부분
- PEACE/PDL로 나타나는 일반적인 프로세스 모델이나, 최적화된 프로세스 모델의 라이브러리
- 프로세스 실행 중에 사용되는 소프트웨어 공학 툴 라이브러리
- 라이브러리를 사용할 때 필요한 관리를 담당하는 부분
- 프로세스 모델의 지식 기반을 다룰 때 필요한 관리를 담당하는 부분
- 사용자에게 보여지게 되는 유저 인터페이스



[그림 2.8.2 : PEACE Environment Architecture]

2.8.4 결론

PEACE 프로젝트를 통해서 소프트웨어 프로세스의 dynamicity 와 evolvability 를 지원할 수 있었다. Goal-oriented 접근 방식을 사용하여 소프트웨어 프로세스의 dynamic한 측면을 다룰 수 있었고, nonmonotonic logic-based formalism을 통해서 소프트웨어 프로세스의 evolution을 가능케 하였다. 따라서, 프로세스의 activity들을 goal 을 달성하는 방향으로 재구성 하거나, 프로세스를 개선 또는 확장할 수 있으며, 프로세스 실행 중에도 일관성을 유지하는 가운데 이러한 기능을 제공해 줄 수 있었다.

2.9 E3 : Object-Oriented Software Process Model

2.9.1 개발배경 및 특징

E3 프로젝트는 process model을 implement하기 전에 process model을 design하기 위한 방법으로서 고안되었으며 OO를 기반으로 한다.

OO design 방법이 연구된 이유는 다음과 같다.

- Process model은 복잡하고 이해하기 어렵다. 따라서 structuring mechanism이 필요하다.
- OO design은 process model을 증명하고, simulation하고, implement하는데 특정 언어를 사용하도록 하는 제약 조건이 없다.
- OO approach들이 software process model programming에 성공적으로 적용되어왔다.

처음에는 object-oriented Coad and Yourdon 방법론이 연구되었고, ISPW6 예제 문제와 local 소프트웨어 조직의 소프트웨어 process 두 가지를 design하는데 사용되었다. 이러한 OO design은 다음 두 가지를 기반으로 이루어져있다.

- PM semantics를 갖는 kernel class들과 relation들의 set
- Domain specific한 class들과 relation들의 set

그런데 PM concern relation들에 Coad and Yourdon 방법론을 적용하면서 여러 가지 문제점들이 생겨났다.

- 제한된 class 간 relation들만을 제공함으로써 designer가 자신의 필요한 새로운 relation을 정의할 수 없었다.
- PM domain에 대해서 잘 정의된 semantics를 갖는 제한된 relation set들을 identify, specify하기 어렵다.
- 규모가 큰 Coad and Yourdon design들은 각 class에 사용된 icon들이 같아서 이해하기 어렵다.

그래서 E3 방법에서는 process model designer나 이 방면에 대한 지식이 없는 사람들이 쉽게 이해할 수 있는 4개의 icon들(4개의 kernel class들: *Task*, *Role*, *Data*, *Tool*)을 제공하는 graphical notation을 제공하게 되었다. 또한 process model design organization과 structuring을 위한 4개의 view들을 제공한다.

E3 방법은 특정 implementation language에 제한이 되지는 않지만 simulation을 위해서

SmallTalk language를 기반으로 하는 간단한 환경에서 simulate되었다. Process model을 design하기 위해 사용된 class들은 SmallTalk class들로 implement되었고, system 기능들(e.g. user interaction)을 제공하기 위해서 여러 가지의 추가적인 class들이 implement되었다. 프로그램의 실행은 process model을 simulation할 수 있었다. Enaction이 아닌 simulation인 이유는 (1) 프로그램의 실행이 하나의 machine에서만 지역화되어서 이루어졌다는 점(single user)과 (2) tool 통합이 이루어지지 않았다는 점 때문이다. Simulation이 아닌 Enaction이 이루어지기 위해서, 여러 class들은 multi-user와 tool 통합을 지원하도록 수정 혹은 개선되어야한다.

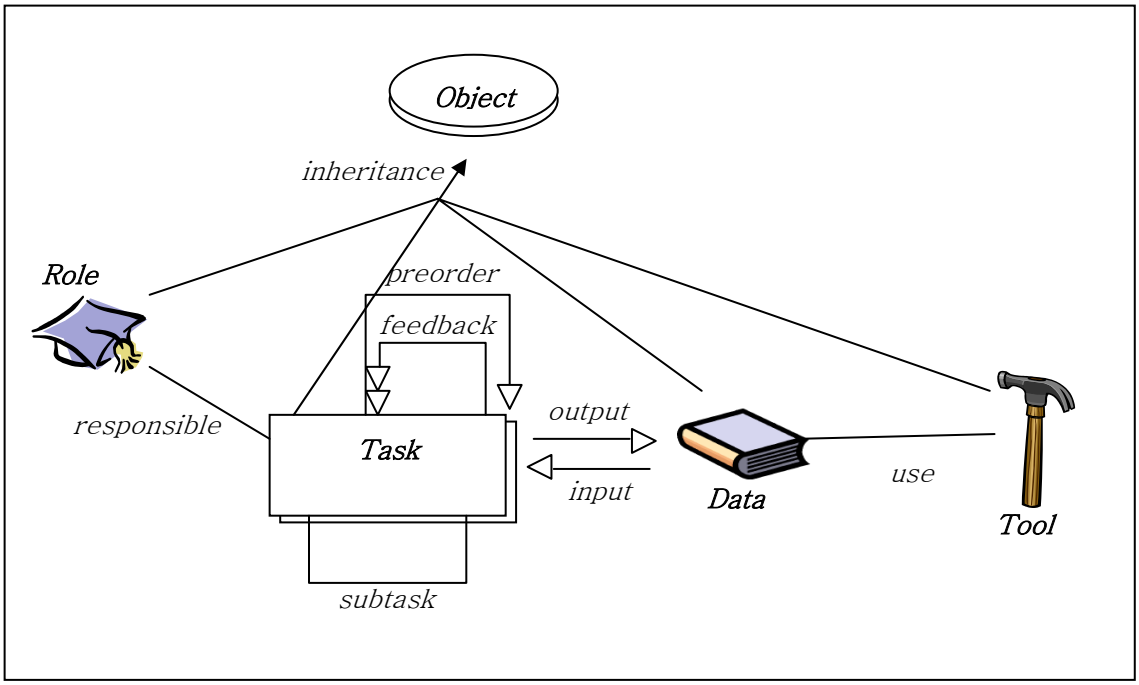
E3 방법은 meta-process model을 design하는 데에도 적합하다. Meta-process에 특화된 엔티티들은 PM의 특화된 class들의 specialization(예를 들면 meta-task들은 *Task*의 specialization으로 모델 가능) 혹은 generic class들(예를 들면 meta-data는 *Object*의 specialization으로 모델 가능)로 모델링될 수 있다.

E3 방법은 모델의 정적인 부분만 표현할 뿐, 동적인 부분들(control, instantiation 등)에 대해서는 지원을 하지 못하기 때문에 불완전하다.

2.9.2 E3 Process Design 방법

2.9.2.1 구조

E3에서의 process design은 kernel class들과 relation들, 4개의 view들로 구성된 presentation rule들로 구성된다. Kernel class들은 모두 class *Object*를 상속 받는다. 각각 activity, role, product, tool submodel의 root가 되는 *Task*, *Role*, *Data*, *Tool*이란 4개의 kernel class들이 있으며, 이들은 모두 abstract로, 직접 instantiate되지는 않는다. 그러나 이들은 더 많은 specialized subclass들을 생성하는데 사용된다. Kernel class들은 PM specific functionality들을 제공하게 된다. User-defined class들은 상속을 통해서 이러한 functionality들을 얻을 수 있다.



[그림 2.9.1 : Kernel classes and relations]

2.9.2.1.1 Kernel Classes

2.9.2.1.1.1 Class *Object*

Object들의 전형적인 OO functionality들을 정의한다. Graphical notation은 타원이다. Task, Tool, Data, Role submodel로 들어갈 수 없는 class들은 Object의 subclass로 선언되면 된다.

2.9.2.1.1.2 Class Task

Software 제작 활동들은 Task를 모델로 한다. Task는 subtask들로 나눠진다. Sibling task들의 수행 순서는 relation feedback과 preorder에 의해서 정해진다. Graphical notation은 shaded box이다. Task는 그의 life cycle동안에 attribute state의 value에 의해 identify되는 여러 개의 state들을 거친다.

Task class는 다음과 같은 여러 operation들을 제공한다.

1) handleCompletion

subtask 또는 sibling task에 의해 불려지며(called), 부른(calling) task state가 수정됐음을 알려준다.

2) startExecution

handleCompletion에서 task 실행을 시작하거나, 재실행하기 위해서 불려진다.

3) restart

feedback relation에 연결된 실패된 task에 의해서 receiver의 재실행을 강제하기 위해서 불려진다.

4) readyToInteract

task에게 user가 interact될 준비가 되었다는 것을 알려주기 위해서 불려진다.

2.9.2.1.1.3 Class Role

human interaction에 필요한 활동들은 연관된 role을 가진 task에 의해서 모델되어야한다. Task는 relation responsible로 수행할 책임이 있는 role과 연결되어야 한다. Graphic notation은 모자이다. Abstract class인 Role은 제공해야 할 서비스에 의해서 특징지어지는 것이 아니고, 그것이 참여하고 있는 relation에 의해서 특징지어진다.

2.9.2.1.1.4 Class Data

Software 개발 활동의 결과로 생긴 부분적인 혹은 완전한 결과물, 즉 product에 해당되는 모델이 Data이다. Graphic notation은 책이다. Method beOperated는 relation use에 연결된 tool이 data에 대해서 활성화되도록 만들어주는 역할을 한다.

2.9.2.1.1.5 Class Tool

여러 개발 활동들은 design tool이나 compiler와 같이 자동화된 tool에 의해서 수행된다. Relation use는 tool과 그들이 작용하는 data를 연결해준다. Graphic notation은 망치이다. Class Tool은 실행하도록 해주는 operation run을 제공한다.

2.9.2.1.2 Kernel Relations

process model design이 class level에서 만들어짐에 따라, 모든 relation들은 class들간의 관계로 나타난다. 그러나 enactable process model을 만들 때에는 class 레벨과 instance level, 두 레벨 모두에서 relation들을 implement할 수 있도록 할 필요가 있다. 각 instance 레벨 relation은, 대응되는 class 레벨의 relation에 의해 연결되는 class들에 속하는 object들을 연결해야만 한다. 하지만 모든 class 레벨의 relation들이 대응되는 instance 레벨의 relation들을 갖는 것은 아니고, 단지 class 레벨만 강조하는 relation들도 있다. Relation cardinality는 relation semantics에 따라서 달라진다.

2.9.2.1.2.1 Relation inheritance

표준 OO inheritance relation과 같다. Graphic notation 역시 OO에서 사용하는 것

과 동일하다. Inheritance는 명확히 class 레벨에서만 중요하다.

2.9.2.1.2.2 Relation subtask

하나의 task는 relation subtask에 의해서 여러 개의 subtask들로 나뉠 수 있다. Graphic notation은 task와 subtask 각각 사이의 non-oriented edge로 표현된다. Supertask는 subtask들 위에 그려짐으로써 supertask와 subtask 사이의 애매함을 없앨 수 있다.

2.9.2.1.2.3 Relation input and output

Task는 input을 가지고 output, class Data (또는 Data의 subclass)의 instance,을 만들어낸다. 모델에서는 이것을 각각 relation input과 output으로 명시할 수 있다. Relation input은 Data subclass에서 Task subclass로의 화살표로 표시된다. Relation output은 Task subclass에서 Data subclass로의 화살표로 표시된다.

2.9.2.1.2.4 Relation preorder

process model을 design할 때, task들 사이의 수행 순서를 명시적으로 표현할 때가 있다. 이때는 relation preorder를 두 개의 Task들 사이에 화살표로 표시해준다. 화살표 이전의 task가 끝나야 화살표 다음의 task가 시작할 수 있다.

2.9.2.1.2.5 Relation feedback

한 task의 실행이 다른 task를 재실행시킬 수 있다. 예를 들면, review 단계에서 관련된 production 단계를 재실행할 수 있다. Relation feedback은 한 task에서 재실행될 수 있는 task로의 double arrowed 화살표로 표시된다. Feedback에 관한 정보는 재실행을 일으키는 task에 의해 생성되는 document에 포함되어있다.

2.9.2.1.2.6 Relation responsible

Relation responsible은 Task subclass와 Role subclass 사이에 edge로 표시된다.

2.9.2.1.2.7 Relation use

주어진 Data를 조작하는 tool이 무엇인지 명세해 준다. Data subclass와 Tool subclass 사이의 edge로 표현된다.

기존의 다른 PM formalism들은 tool들과 사용되는 task들을 연관지어 주지만 여기에는 다음과 같은 문제점들이 있다.

1) 하나의 activity를 수행할 때, 두 개 이상의 tool들이 필요한 경우가 있다. 예를 들면, 수정 작업은 소스 파일의 코드를 수정하기 위한 editor와 design

document들을 접근하기 위한 design tool이 모두 필요하다. tool들과 task들을 연관 지어주면 어느 tool이 각 input data와 output data에 작용되는지 알 수가 없다.

2) 하나의 새로운 data가 하나의 task에 연관될 때마다, 이 data를 조작하기 위한 tool 역시 task에 연관되어야 한다. 만약 이러한 relation이 없다면, 잘못된 process model이 나오게 될 것이다.

3) 몇몇 tool들은 다른 어떠한 tool로도 읽을 수 없는 format으로 data를 저장할 수 있다. 따라서 이러한 data를 model하고 있는 Data subclass의 instance들은 그들의 creator tool과 연관 지어져야 한다.

앞의 1), 2) 문제점들은 process modeling에 관한 관점으로서 class level의 relation use로 해결 가능하다.

3) 문제점은 process model execution에 관한 관점으로서 instance level의 relation use로 해결 가능하다.

2.9.2.1.3 Presentation Rules

E3 process model design의 graphical 표현은 4가지 view들로 이루어져있다. 각 view들은 다음과 같다.

1) Inheritance View

주어진 모델의 inheritance hierarchy를 나타낸다. 따라서 class들은 표준 OO inheritance relation으로 연결된다.

2) Task View

하나의 task 기능들의 순간적인 one-level view를 보여준다.

3) Task Decomposition View

하나의 task를 subtask relation에 의해 subtask들로 나눈 two-level view를 보여준다. subtask간의 연결 관계를 제어하는 것은 feedback relation과 preorder relation에 의해서 이루어진다. Task breakdown에서 주어진 task가 차지하는 위치는 path를 이용해서 추적이 가능하다. Path는 Root task에서 시작하여 모델 전체의 process에 걸쳐있으며, 이를 이용하여 traceability를 잃지 않고, modularization을 얻을 수 있다.

4) User View

User-defined class들과 relation들을 표현한다. User View를 나타내는 특별한 rule은 정해져 있지 않으며, 이는 process model designer의 책임이다.

2.9.2.2 예제

간단한 process model을 design하기 위한 예제의 informal specification은 다음과 같

다.

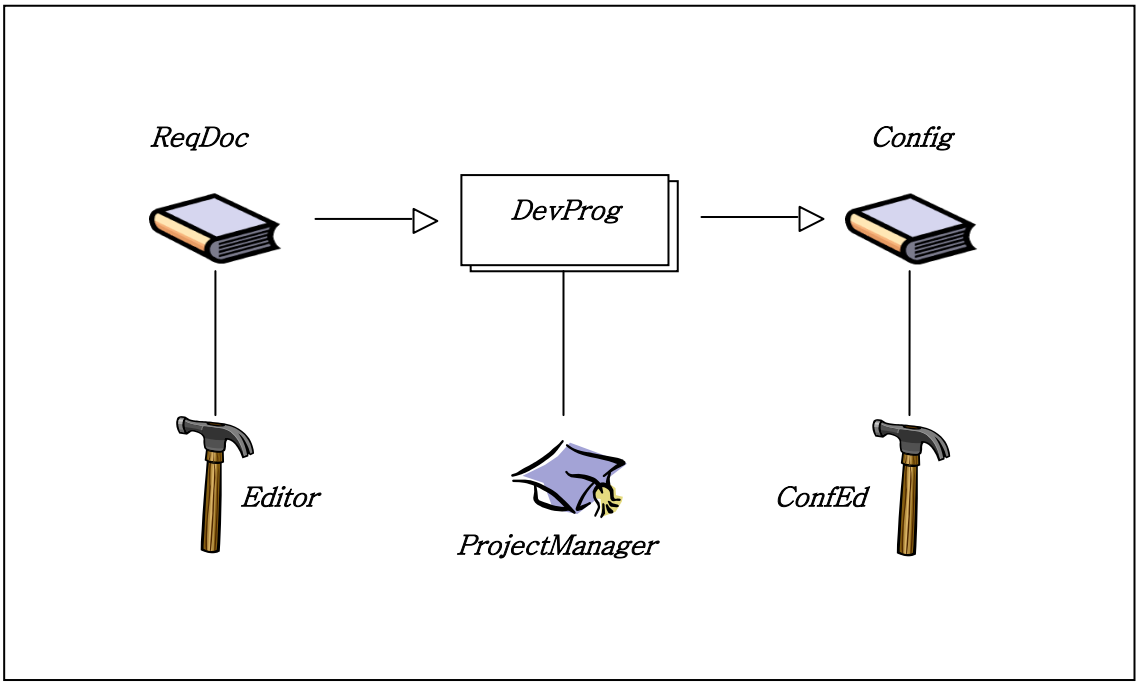
Software process는 design, review design, program으로 구성되어있다.

개발자 팀 내에서는 한 명의 Project Manager가 있고, 그는 designer들과 reviewer들, programmer들을 task들에 할당하는 책임과 사용할 design tool을 선택할 책임이 있다.

Editor는 user에 의해서 선택된다.

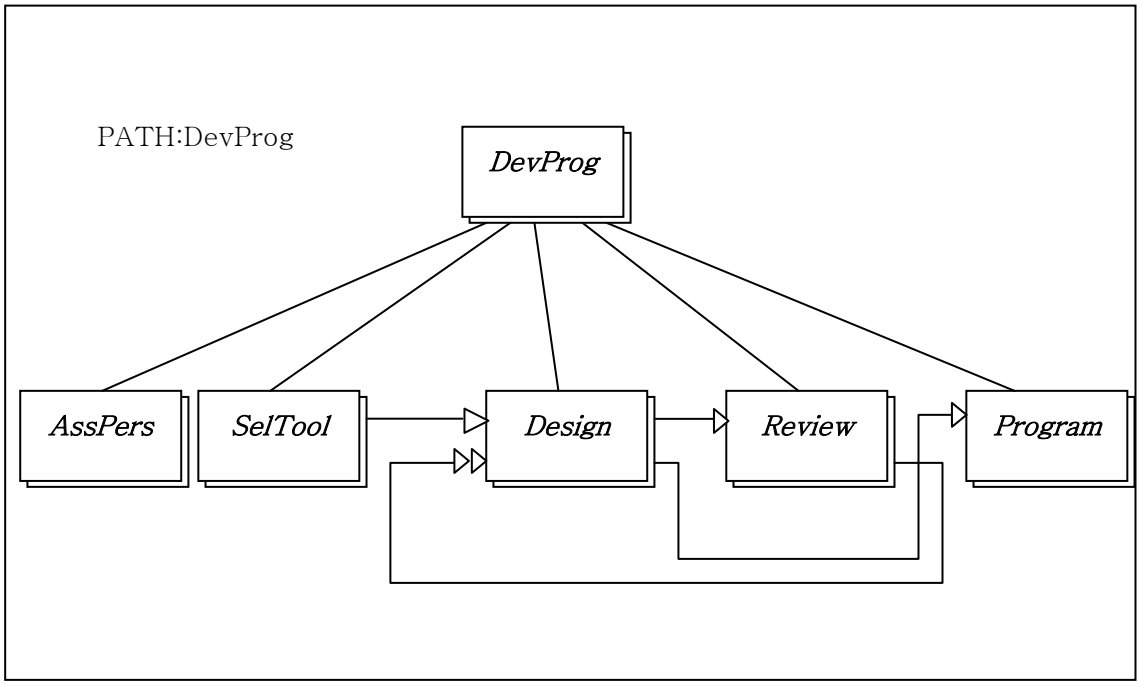
Review와 program 단계는 동시에 시작될 수 있다. 그러나 product는 두 단계가 모두 끝나야 만이 deliver된다.

모든 process는 task DevProg로 모델 된다. DevProg는 input으로 ReqDoc의 instance를 가지고, output으로 configuration을 생성하며, 이는 ProjectManager의 책임이다.



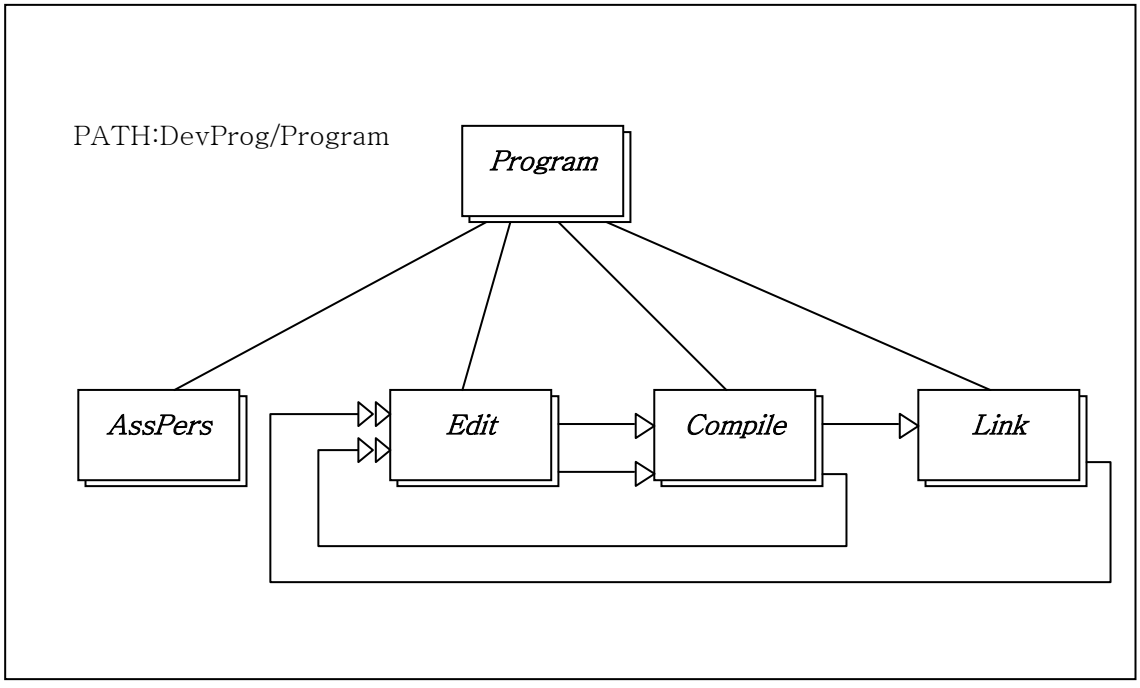
[그림 2.9.2 : Task View of DevProg]

DevProg의 Task Decomposition View는 다음과 같다.



[그림 2.9.3 : Task Decomposition View of DevProg]

Program은 다시 Edit, Compile, Link, AssPers으로 나뉘어진다.

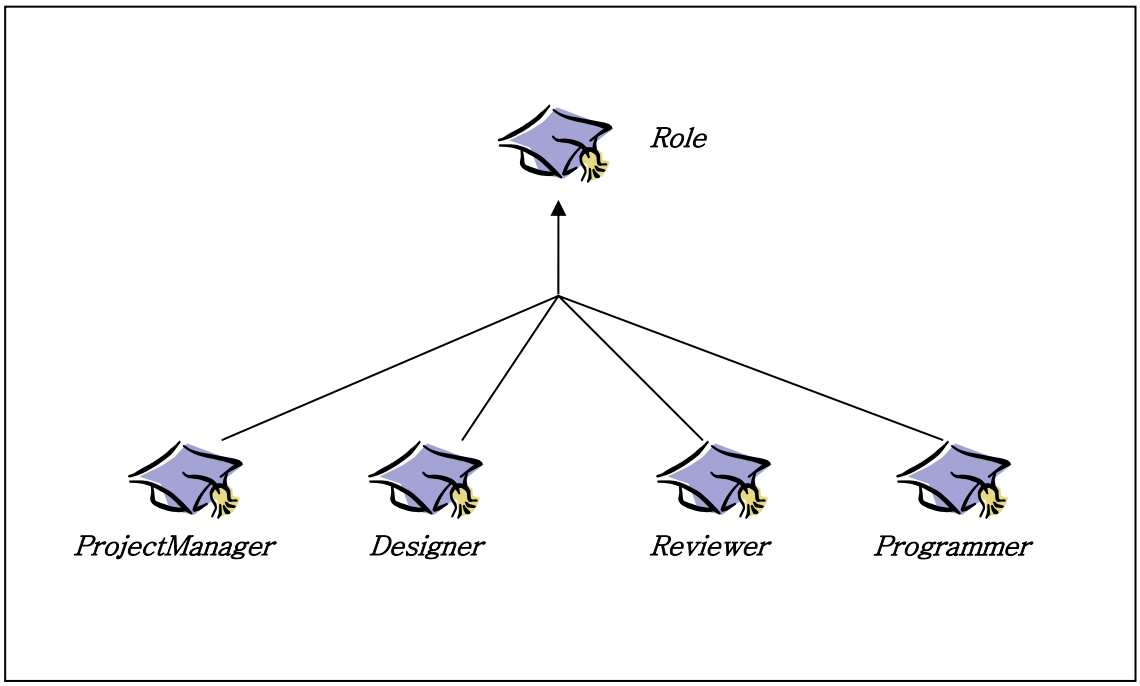


[그림 2.9.4 : Task Decomposition View of Program]

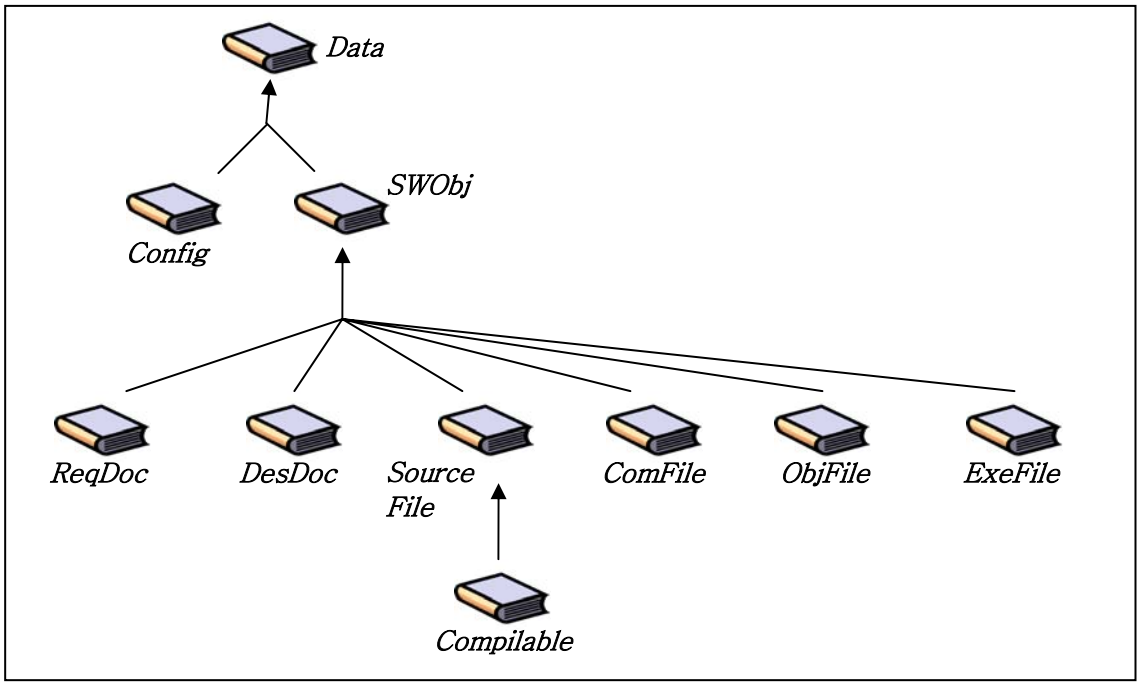
AssPers는 project management facility이다. AssPers가 활성화되면 사람들을 subtask

들로 할당하는 역할을 맡은 사람과 상호작용하게 된다. Allocation의 책임을 갖는 사람의 역할은 model level에서 결정되며, 그 역할은 Project Manager 자신이 직접 할 수도 있고, 아닐 수도 있다.

Role과 Data의 Inheritance View는 각각 다음과 같다.



[그림 2.9.5 : The Role Inheritance View]



[그림 2.9.6 : The Data Inheritance View]

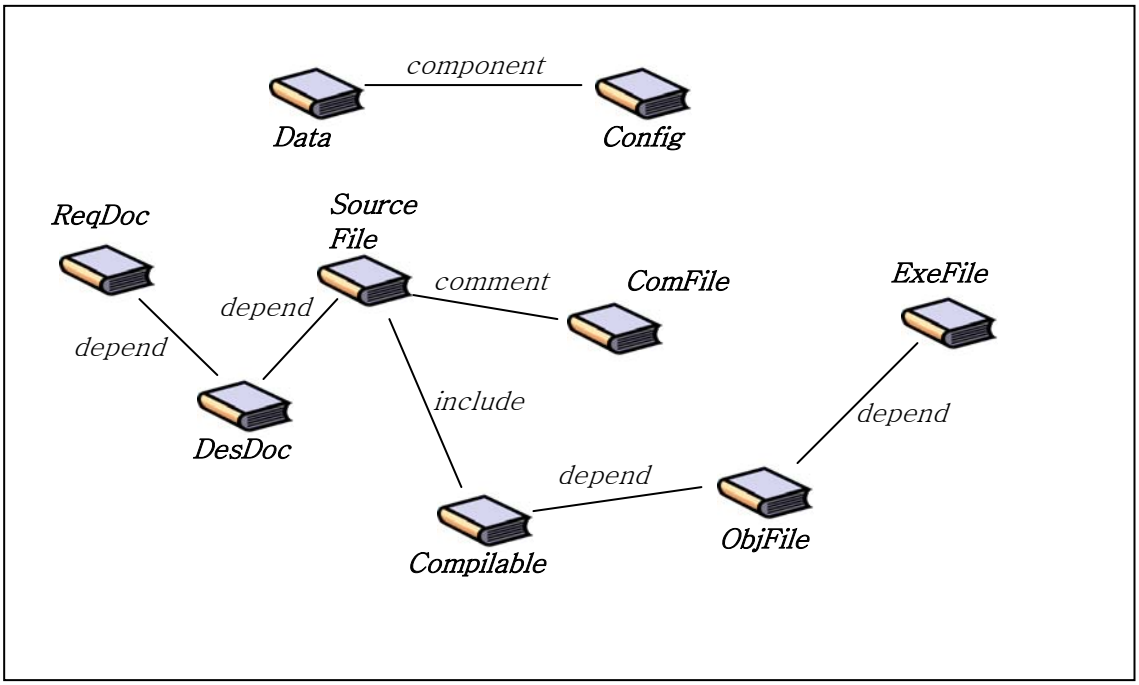
그림 2.9.7은 Data subclass들에 대한 User View를 보여준다. 이 view는 product configuration 스키마를 보여준다.

Config는 Data를 컴포넌트로 하는 product configuration을 모델한다. Relation component는 하나의 configuration과 그의 컴포넌트들을 연결해준다.

Class level의 relation depend는 configuration 스키마에서 dependency를 명세 시켜준다. Instance level의 relation depend는 한 document와 그 document로부터 나오는 다른 document들을 연결해준다. 예를 들어, ReqDoc은 requirement document를 모델하고, DesDoc는 design document를 모델한다. 그런데 depend는 이들 두 종류 document 사이의 dependency를 설정해준다.

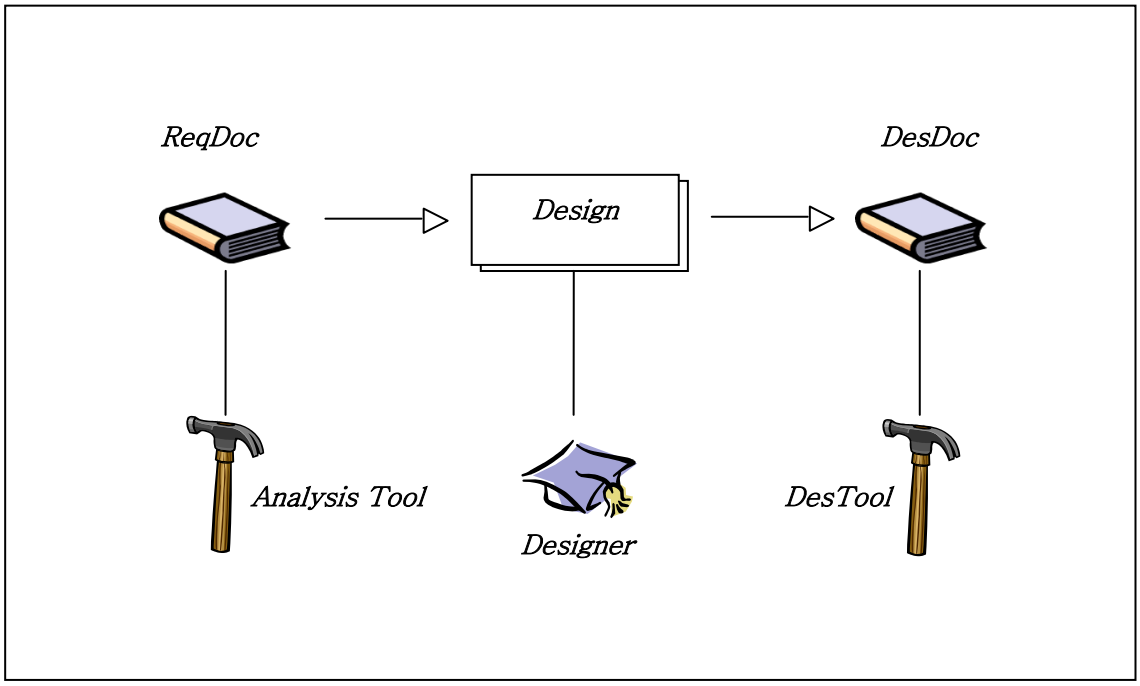
Class Compilable은 SrcFile의 subclass이다. 이것은 컴파일될 수 있는 소스 파일들을 가리킨다. 하나의 compilable 파일은 relation use로 연결된 여러 소스 파일들을 포함할 수 있다. 하나의 소스 파일은 relation comment에 의해 연결된 ComFile class의 instance에 의해 표현되는 comment 파일과 연결될 수 있다.

Class ObjFile은 relation depend에 의해 연결된 소스파일들을 컴파일함으로써 얻어지는 object 파일들을 모델한다. 컴파일은 relation include로 연결된 object들을 포함시킨다. ExeFile class의 instance들은 depend로 연결된 object파일들을 링크시킴으로써 생겨나는 실행가능한 파일들을 표현한다.



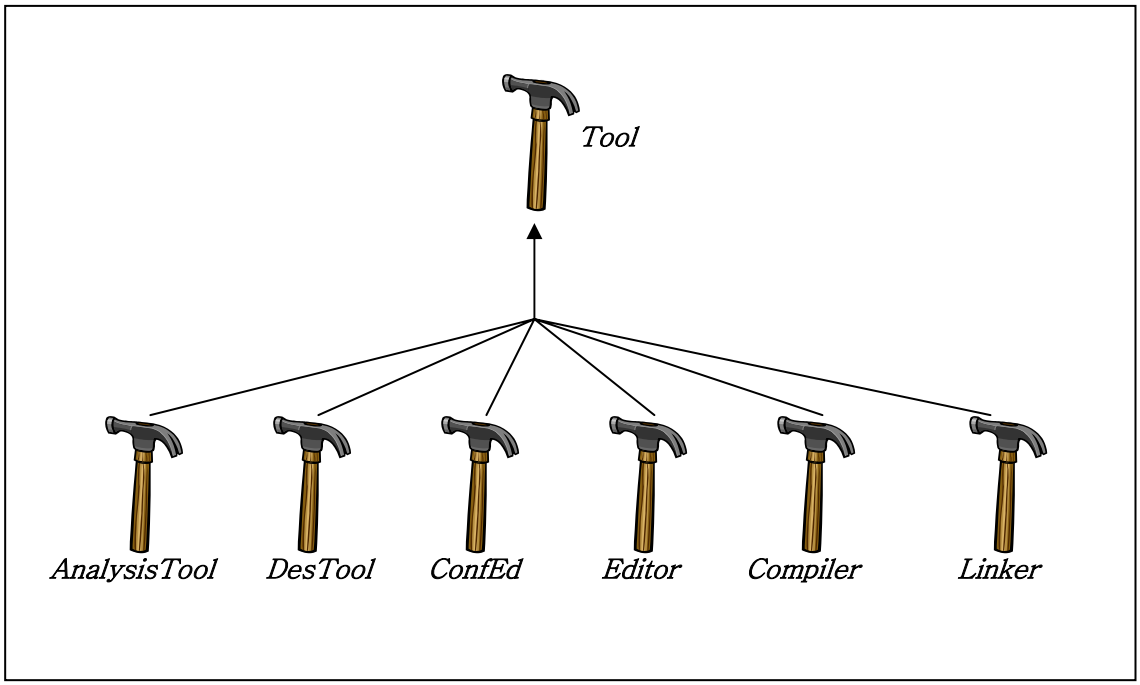
[그림 2.9.7 : The Data User View]

그림 2.9.8은 Design의 Task View이다. 이 그림에서 역시 ReqDoc와 DesDoc 사이의 dependency를 볼 수 있다. Task Design은 ReqDoc의 하나 이상의 instance들을 input으로 가지고, output으로 DesDoc의 instance를 생성한다.



[그림 2.9.8 : The Design Task View]

다음은 Tool의 Inheritance View를 나타낸다.



[그림 2.9.9 : The Tool Inheritance View]

2.10 PADM : Towards a Total Process Modeling System

2.10.1 개발배경 및 특징

PADM은 'Process Modeling Cookbook'에서 제안되었으며, 프로세스 모델링 뿐만 아니라 Socio-Technical 설계, Soft System 개발방법론, 조직 이론 등의 분야에 이론적 기초가 되고 있고, Manchester대학의 Information Process Group (IPG)의 사례연구를 통한 action research에 부분적인 기초가 되고 있다. PADM은 소프트웨어 모델의 획득, 모델링, 실행 등을 포함한 개발중인 방법론으로써, 실행 시스템과의 접목을 시도하고 있다. 현재 PADM은 실행 시스템으로 ICL의 ProcessWise Integrator (PW Int) 제품을 사용하고 있다. PADM이 실행 시스템과의 접목하는데 있어서, IPSE 2.5 project의 선행 연구에서 소개된 Base Model (BM)과 PW Int의 두 가지 기술이 중요하다.

2.10.2 Base Model

2.10.2.1 소개

BM은 IPG가 개발한 프로세스 모델링을 위한 명세 방법이다. BM이 프레임워크를 제공하여야 하기 때문에 명세는 체계적인 방법으로 쓰여진다. 따라서, 각 명세 construct는 논증 가능한 temporal logic semantic으로 주어진다. 소프트웨어 프로세스의 실행 가능한 모델을 개발하는 노력에 있어, BM 명세를 실행할 수 있는 도구인 'BM Stepper'가 개발되었다.

2.10.2.2 개요

BM에서, 시스템은 병렬적으로 실행되는 컴포넌트로 구성된다. 각 컴포넌트는 많은 operation을 제공하고 operation에 접근할 수 있도록 state variables를 포함한다. 하나의 operation은 다른 컴포넌트의 다른 operation을 호출할 수 있다. Operation에는 active와 passive 두 종류가 있다. Active operation은 컴포넌트 자체에서 실행되고 passive operation은 다른 컴포넌트에서 다른 operation에 의해 호출 될 때에만 시작된다. 컴포넌트의 행위는 컴포넌트가 실행될 때, 그 operation이 따르는 패턴에 의해 명세된다. 그러한 패턴을 operation pattern이라 부른다. 컴포넌트의 실행은 다음과 같다. Operation pattern에서 다음 operation이 active operation이면, 외부 호출을 기다리지 않고 실행될 수 있다. 만약 passive operation이라면, 컴포넌트는 다른 컴포넌트의 operation에 의해 호출될 때까지 기다려야만 한다. 한 컴포넌트는 다른 컴포넌트와 operation의 호출을 통해서만 통신할 수 있다.

2.10.3 ProcessWise Integrator

2.10.3.1 소개

ICL ProcessWise는 다각적으로 조직적인 프로세스 활동을 지원하기 위해서 세 가지 서비스를 제공한다. 'Business Modeller's Workbench'는 프로세스 행위를 모델링하고 시뮬레이션 하는 도구를 제공하고, 'Integrator'는 프로세스 모델링, 통합, 실행 시스템이다 'Guide'는 앞의 두 가지 기술들을 사용하기 위해 고안된 특정한 개발방법론이다. 이 특정 개발방법론이 ProcessWise Integrator (PW Int)이다. 이 시스템은 변환 사건(transformation events)을 위한 메커니즘을 정의하고 이의 실행을 위해, 역할/활동/상호작용의 개념을 취한다. 결국 PW Int는 정보 시스템 개발 방법론의 프로세스를 표현하는데 있어, 도구들이 효과적으로 조합될 수 있는 프레임워크를 제공한다.

2.10.3.2 The Process Programming Language (PML)

PML은 PW Int에서 중요하며, 그것은 IPSE 2.5 프로세스 모델링 언어에서 비롯되었고 아직 그 특성의 대부분을 포함한다. 다음은 PML의 6가지 특성을 보여준다.

- Concurrent threads of execution : Threads는 프로그램 디자인 태스크(task)를 단순화시키는 편리한 추상화(abstraction)를 제공한다.
- Dynamic thread creation : 새로운 thread의 동적 생성은 새로운 활동이 하나의 프로세스 안에서 시작될 수 있도록 한다.
- Subtyping : Subtyping은 operation이 관련 있는 다른 데이터 오브젝트에 적용될 수 있도록 정의되는 것을 허용한다.
- Persistence : PW Int 환경에서 제공하는 persistence는 프로그래머에게 프로그램 데이터가 어느 기억 장치에 저장되어 있는지 고려하지 않게 한다.
- User Interface : PML은 사용자에게 프로세스 thread의 상태와 프로세스의 owner가 보는 display간의 단순한 관계를 정의하도록 한다.
- Application Interface : 시작하는 능력과 PW Int 밖의 데이터를 애플리케이션으로 전송하고 전송 받는 능력은 프로세스 프로그램에게 프로세스 상태와 그들의 operation이 조화하여 움직일 수 있게 한다.

2.10.3.3 The Type System

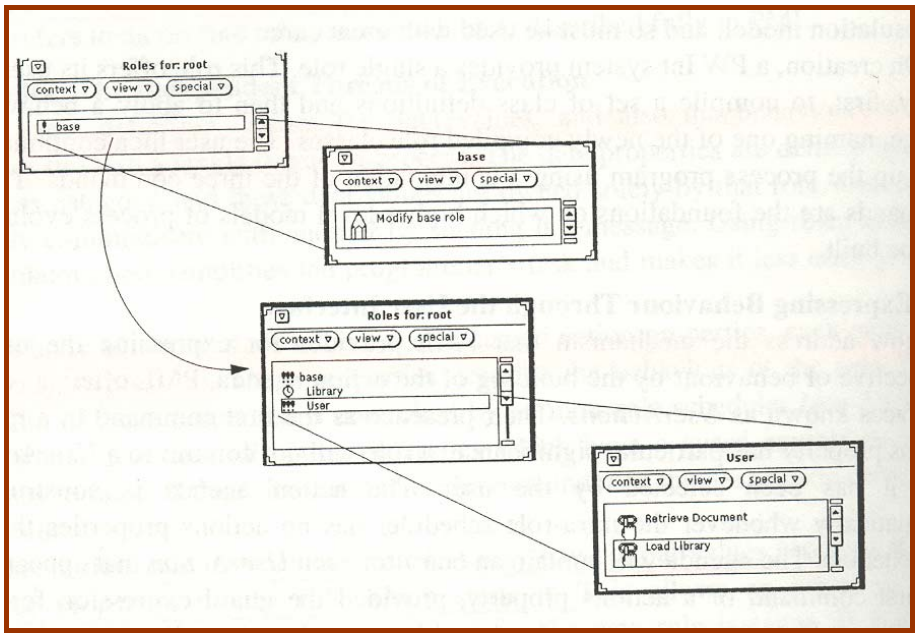
PML은 클래스 기반의 언어이다. 언어가 개발된 초기에는 type-checking이 가능하지 않았지만. 현재는 클래스 상속 구조를 유지하면서 동시에 type-checking이 가능하게 되었다. 클래스 계층은 entities, actions, roles의 클래스를 지원한다. Entity 클래스 정의는 record type을 생성하고, action 클래스 정의는 프로시저를 소개하고 role 클래스 정의는 계속되는 실행 thread 생성을 위한 schema를 제공한다.

Role 클래스 정의는 데이터 값과 프로세스의 한 thread를 통해 생성된 instance의 행위를 정의한다. 데이터 속성은 resource category에 저장되어 있으며, 이 데이터 값은 하나의 role이 단독 소유한다. 하나의 role은 다른 role과 메시지를 주고 받으면서 통신한다. 이렇게 role을 데이터를 캡슐화에 사용함으로써 프로그래머의 일을 단순화시키고, 에러가 발생할 확률을 줄인다. Role의 body는 '*when*'으로 표현되는 named action properties으로 나뉘어진다. 이 properties는 role의 행위를 묘사하고, properties의 실행은 intra-role scheduler에 의해 제어된다. 첫째, intra-role scheduler는 false인 guard expresseion을 포함한 모든 action properties를 줄이고, 남아 있는 action properties 중에서 다음에 실행할 것을 선택한다. 새로운 role instance가 생성되면, 그 role을 생성한 role에 의해 propertise가 주어진다. Role간의 통신은 message-passing을 통해 이루어지며, 통신 메커니즘은 비동기식이다. Giver는 taker를 위해 기다리지 않는다. 그러나 receiver는 데이터를 보내기 위해서 기다려야 한다. PML은 프로그래머가 이렇게 되도록 notation을 제공한다. 이러한 role간의 통신 메커니즘은 subtyping을 이용할 수 있다. PML은 cast operation을 제공하여 손실된 정보가 다시 추출될 수 있도록 하며 run time type check도 가능하게 한다.

PML은 프로세스 프로그램이 진화할 수 있도록 세 가지 언어 인터페이스를 제공한다. 그 하나는 새로운 실행 thread를 생성하는 것이며, 두 번째는 PML 컴파일러의 인터페이스이고, 이것은 클래스 정의가 담긴 소스를 받아 컴파일된 클래스 값으로 되돌려준다. 세 번째 인터페이스는 role instance가 새로운 클래스 정의에서 재정의 될 수 있도록 한다. Role instance의 데이터는 공유할 수 없기 때문에, 재정의된 role이 전의 상태를 유지할 수 있도록 한다. 이 인터페이스는 데이터 캡슐화된 모델을 깨기 때문에 충분한 고려 후에 사용하여야 한다. PW Int 시스템에서는 하나의 role을 제공하고 이 role은 사용자에게 첫째, 클래스 정의를 컴파일 할 수 있도록 하고, 새롭게 컴파일된 role 클래스에게 이름을 부여함으로써 변화를 적용할 수 있도록 한다. 사용자는 계속해서 세 가지 언어 인터페이스를 통해서 프로세스 프로그램을 만들어 낼 수 있다.

PML은 사용자의 행위를 표현하기 위해 action agenda를 제공한다. PML은 *UserActions*인 인터페이스의 집합을 제공한다. 이것은 role의 action property에서 처음 나타나는 명령어로써, 중요성을 지닌다. Role은 사용자가 선택하기 전까지 *UserAction*에 commit하지 않는다. Action agenda는 intra-role scheduler가 schedule할 어떠한 action properties도 갖지 않을 때 마다 자동적으로 생성된다. Agenda는 property에 대한 guard expression이 true면, actions property에서 첫 명령어로 나타나는 각 *UserAction*에 대한 entry를 포함한다. 일단 사용자가 action을 선택하면, action이 취하는 절차에 따라 스케쥴된다. *UserAcion*을 스케쥴링하는 것은 UI object를 생성한다. PW Int UI

model은 계층적이다. 가장 상위 레벨에서, 참여자 role이 role agenda로 수집된다. 각 role은 action agenda를 펼친다. 이것은 참여자에게 그 role에서 현재 가능한 사용자의 action 목록을 보여준다. 참여자는 그 중 하나를 선택하고, 선택된 action은 그에 맞는 형태의 윈도우를 띄운다. 예를 들어, 이것은 문서 편집 윈도우이거나 공란을 채우는 form형식일 수 있다.



[그림 10.1 : User Interface]

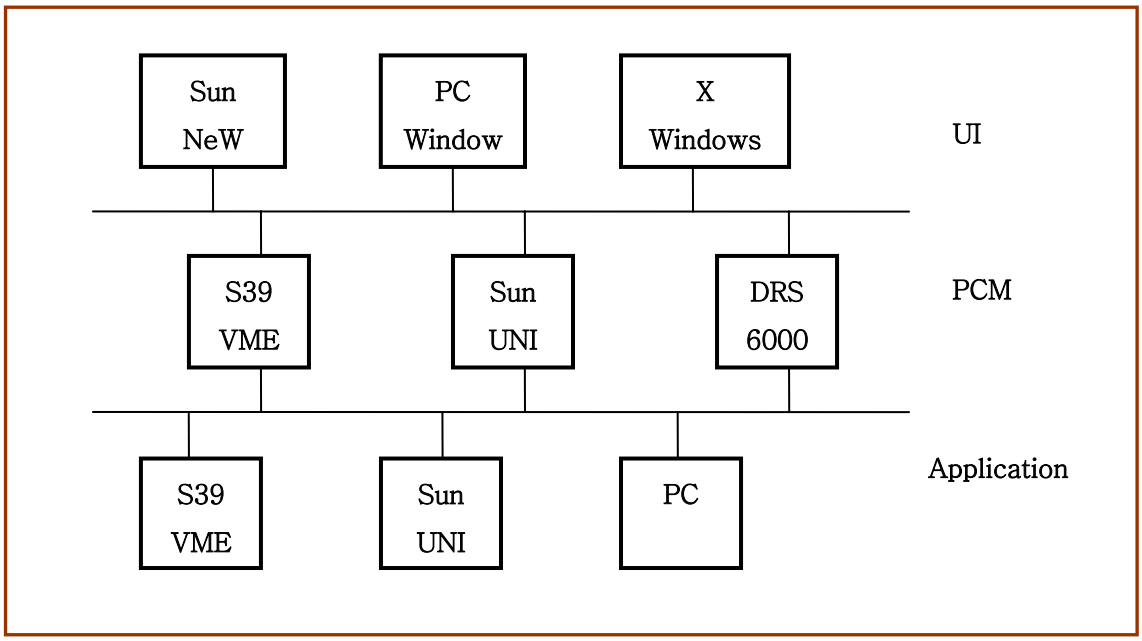
위의 그림은 시스템이 단순한 사용자 클라이언트 서버 애플리케이션을 구동했을 때 UI 윈도우의 모습이다.

PML은 프로세스 프로그램이 PW Int 시스템 외부에서 돌아가는 애플리케이션을 제어할 수 있는 기본적인 능력을 제공한다. 통신 채널은 외부 애플리케이션과 연결해주는 데 사용되며, 각 애플리케이션은 통신 채널의 하나의 parcel로서 여겨진다. 이러한 형태는 초기에 프로세스 프로그램 외부에서 정의된다. 일단 애플리케이션을 관리하는 애플리케이션 서버가 PW Int에 로그인 하면, 프로세스 프로그램에서 사용할 수 있다. 그런 다음 프로세스 프로그램은 서버에게 통신 채널의 parcel instance를 생성하도록 지시한다. 애플리케이션 서버는 관리하는 애플리케이션과 같은 환경에서 실행되도록 작성된 프로그램이다. PW Int 시스템에서는 코드상에서 애플리케이션과 프로세스 프로그램간의 translation 서비스를 제공한다.

2.10.4 도구

PW Int 시스템은 세 가지 아키텍처 요소 - Process Contorl Magnager (PCM)이라 부르

는 프로세스 엔진과 UI 서버, 애플리케이션 서버 - 로 이루어져 있다. UI 서버의 인스턴스는 각 참여자의 워크스테이션에서 돌아가고, 로그인을 통해 각 서버를 PCM로 연결시킨다. 하이레벨 프로토콜이 워크스테이션의 디스플레이 내용과 참여자의 행위를 PCM으로 보내는 통신이 가능하도록 사용된다. 비슷한 로그인 프로시저가 애플리케이션 서버에 의해서도 수행된다. 애플리케이션 서버는 애플리케이션에 적절한 환경하에서 실행된다.

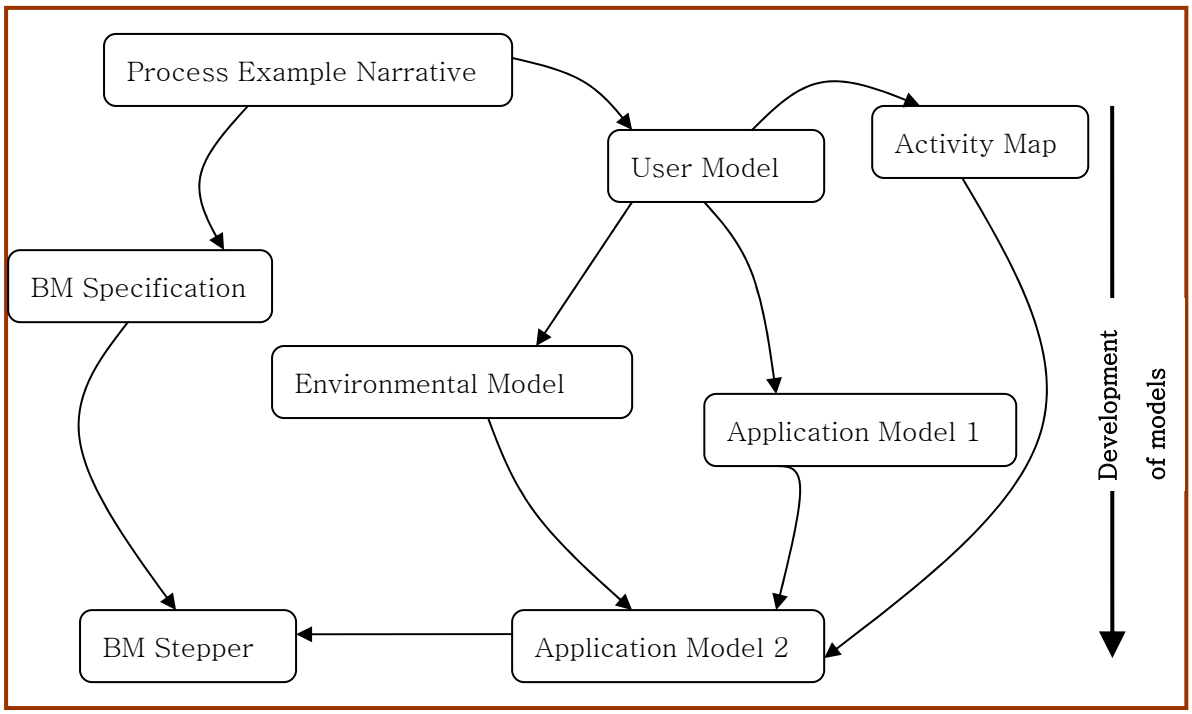


[그림 10.2 : PCM과 서버 환경]

PCM 구현은 SUN3, SUN4, DRS 6000과 ICL 39 시리즈를 위해 존재하고, UI 서버는 NeWS와 SUN workstation에서 돌아가는 X Windows와 PC에서 돌아가는 Microsoft Windows를 위해 쓰여진다. Tool 서버는 SUN 3, SUN4, PC와 DRS 6000 환경을 위해 존재하고, 39 시리즈에 VME 애플리케이션은 Unix tool 서버를 통해 호출될 수 있다. 표준 통신 프로토콜이 서버와 PCM을 연결하기 위해 사용된다. 기계어에 가까운 언어들이 inter-machine 통신을 다루기 위해 사용되지만, PCM은 주로 PS-algol로 쓰여졌다. 이것은 영속성뿐만 아니라 PS-algol 시스템은 PS-algol 프로그램의 컴파일러에 대한 인터페이스를 제공한다. 성공적인 컴파일이 수행되면, 실행 가능한 프로시저 값을 반환한다.

2.10.5 사례연구

제 6회 International Software Process Workshop에서 개발된 예제 프로세스는 소프트웨어 시스템의 개발, 유지보수와 관련된 조직에서 공통적인 프로세스 중 하나이다. 그것은 하나의 소프트웨어 시스템을 구성하는 특정 모듈이나 단위의 변화와 관련된 활동에 초점을 맞춘다. 다음 그림은 사례연구에 관한 메타 프로세스를 보여준다.



[그림 10.3 : 사례연구의 메타프로세스]

2.10.5.1 Specifying ISPW-6 in BM

ISPW-6 예제 프로세스는 agents, steps 그리고 Inputs/Outputs 의 세가지 요소로 구성되어있다. Agents는 프로젝트 매니저, 디자인 엔지니어, QA 엔지니어 등의 사람들이고, steps는 agent에 의해서 수행되는 행위이다. 예를 들어, 디자인을 수정하는 것, 디자인을 검토하는 것 등이 있다. 하나의 agent가 여러 steps을 수행할 수도 있고, 하나의 step이 여러 agent들에 의해 수행될 수도 있다. Inputs/Outputs는 각 step에서 요구하거나 생산되는 데이터이다. 예를 들어, 현재 디자인, 디자인 검토 feedback과 수정된 디자인 등이 있다. 이러한 ISPW-6 예제 프로세스는 하나의 composed object로 표현되고, 각 agent는 하나의 single object로 표현된다. 이러한 object를 agent object라고 부른다. 각 step은 또한 하나의 single object로 표현되고 이러한 object를 step object라고 부른다. 하나의 agent objects가 특정 step object에 접근하려면 특정 role을 획득하여야 한다. Agent object는 여러 다른 step object에 접근할 수 있으며, step object는 여러 다른 agent object에 의해 접근될 수 있다. 전자는 agent가 다른 role을 취할 수 있도록 하고, 후자는 팀이 일할 수 있도록 한다. 이렇듯 ISPW-6 예제 프로세스는 모든 agent object와 step object를 포함한 composed object로 표현된다.

Agent object는 어떤 단계에 artifact를 전달하는 operation, 어떤 단계로부터 artifact를 건네받는 operation과 어떤 단계를 수행하는 operation을 갖는다. 다음은 각 operation을 모델링 한 것이다. 각 agent object는 그 agent의 operation 순서에 제약을 가하는 operation pattern을 가져야 한다.

supply operation: delivers artifacts to a step by calling the receive operation of the appropriate step object. A supply operation of an agent object may be modeled as a passive operation, i.e. being invoked by an agent, or as an active operation

receive operation : receives artifacts from a step. These operations are called by the supply operations of the appropriate step or agent objects. Receive operations are modeled as passive operations.

step interface operation : calls the corresponding step object to perform (part of) the step. Step interface operations are invoked by the relevant agents. They are also passive operations.

Step object는 어떤 agent object 혹은 다른 step object 혹은 둘 다로부터 step을 수행하기 위한 어떤 artifact(input)을 얻는 데 사용된다. 단계가 완료되면, 어떤 artifact(output)은 다른 agent object혹 다른 step object 혹은 둘 다에게로 전달된다. 그러므로 step object 역시 agent obeject와 같은 세 가지 operation을 제공한다. 각 step object 역시 그 step의 operation 순서에 제약을 가하는 operation pattern을 가져야 한다.

receive operation : receives the input data from either an agent or a step object. These operations are called by the supply operations of the appropriate agent or step objects. Receive operations are passive operations

supply operation: delivers the output data to either an agent or a step object. These operations call the relevant agent or step object. A supply operation of a step object is active because the operation is started by the step object itself when it is ready to deliver data

step interface operation : the step. This operation is called by the relevant agent objects. Step operations are passive operations.

다음은 BM을 사용하여 소프트웨어 프로세스를 모델링 하는 가이드라인이다.

- agent object를 확인
- step object를 확인
- 각 agent object에 대해,
  - supply operation을 확인
  - receive operation을 확인
  - step interface operation을 확인
  - agent object에 대해 operation pattern을 정의
- 각 step object에 대해,
  - receive operation을 확인
  - supply operation을 확인
  - step operation에 명명
  - step operation에 대해 operation pattern을 정의

2.10.5.2 Implementing ISPW-6 in PML

1) 개요

프로세스 활동을 표현하기 위해서 다음 다섯 가지의 submodel로 실용적인 접근방법을 채택하였다.

- Activity model : 이것은 실행 가능한 모델이다
- User model : 시스템이 프로세스 참가자들에게 제공하는 서비스를 묘사하고, 그에 상응하는 책임을 부여한 모델

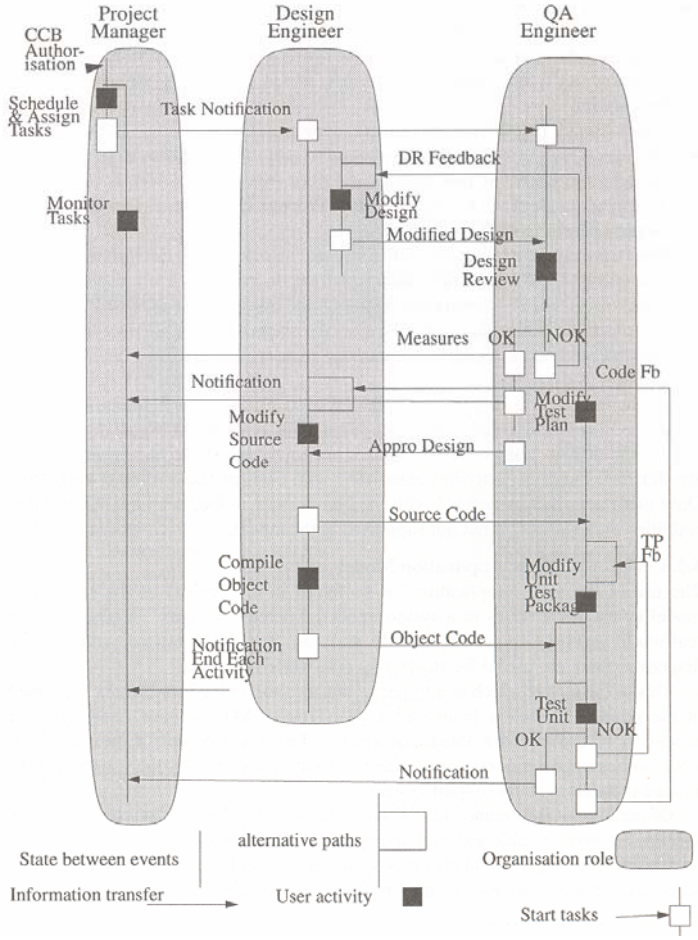
- Organization model : 조직 역할 구조와 이 역할들간에 기능적 관계에 관한 모델
- Product model : 데이터 구조와 관계
- Tool model : 도구 아키텍처의 설명

2) Method와 Tools

PADM에서 표현 method로 사용하는 것은, 첫째, 사용자 role을 확인하여 이 role에 task나 activity를 부여하고 관련된 event을 확인하기 위한 Role Activity Diagram (RAD)이다. 두 번째로 role에 독립적으로 activity를 부여하기 위해 사용되는 Activity Diagram (AD)이다. Dependency Diagram (DD)은 복잡한 프로세스의 본질에 대해서 high level 추상화를 제공한다.

3) The User Model

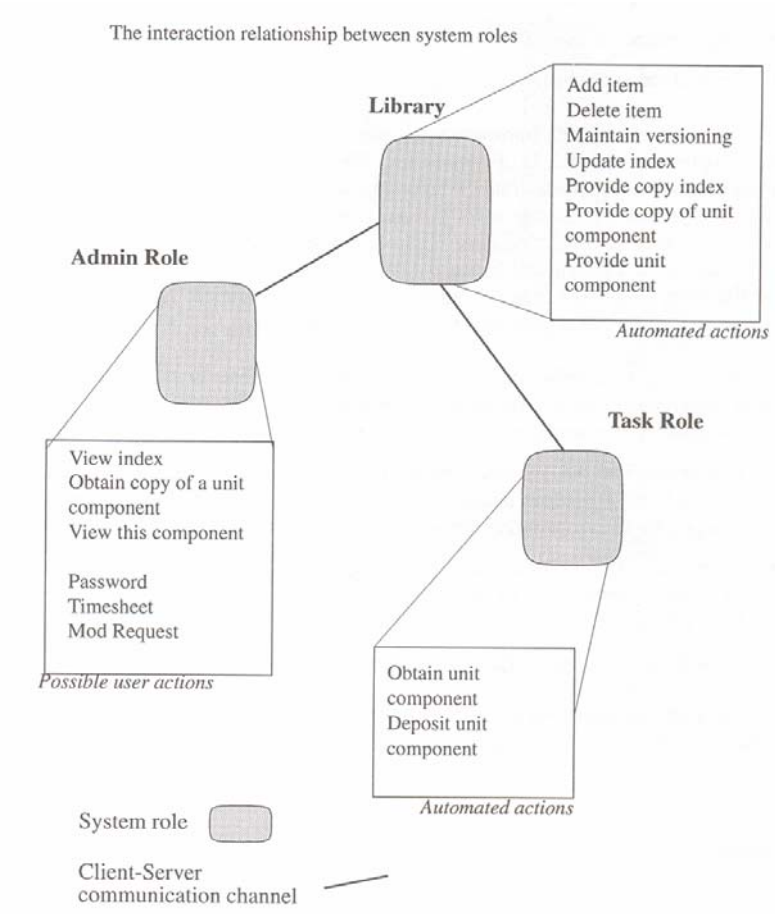
User model은 두 가지 형태를 취한다. Role, tasks와 artifact의 RAD 모델이 그 첫 번째 형태이고, 구현하는데 필요한 더 세부적인 내용은 훨씬 더 잘 정제된 두 번째 형태에서 나타난다. Role Activity Diagram은 role behavior의 범주를 정의한다. 조직에서 role을 고려할 때, RAD는 일어날 수 있는 activity와 activity간의 dependency를 지시한다.



[그림 10.4 : The User Model]

4) The Environment Model

잘 정의된 프로세스가 실행으로서 의미를 갖기 위해서, 프로세스가 실행되기 위한 support나 infrastructure에 주목해야 한다. Environmental aspect를 포함하기 위해 library service가 선택될 수 있다. 이것은 프로세스 예제에서 제공되는 것보다 정보를 제어하는 데 있어 더 적절할 솔루션을 제공한다. 이것은 또한 클라이언트-서버 구조에서 시스템 role이 어떻게 사용되는지 보여준다. 프로세스 실행 시스템이 효과적이기 위해서는 데이터를 가지기 위해 사용되는 library와 상호작용 해야 한다.

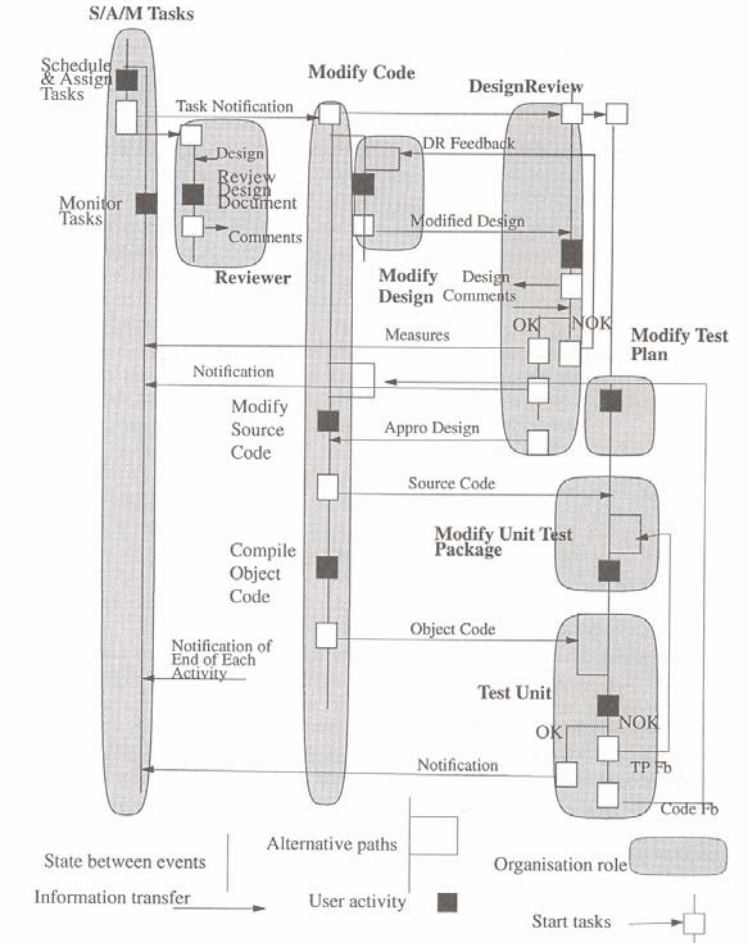


[그림 10.5 : The Environment Model]

이 그림에서, 사용자 시스템 role과 library service role간의 통신 link가 묘사되어 있고, 각 role들과 관련된 activity도 명세 되어 있다. 사실상, library와 관련된 activity만 모델링 될 수 있고, 다른 activity들은 표현되긴 하지만 모델링 되지 않는다.

5) The Activity or Application Model

Application model이란 용어는 보통 system model을 지칭하기 때문에 실행 가능한 모델이란 의미에서 주로 쓰이고, activity model은 실 세계 activity 혹은 시스템 activity를 의미하거나 실 세계 activities를 구조화하기 위해 사용되는 AD와 혼동되어 쓰이기도 한다. 이것은 즉시 실행되는 모델이고, ProcessWise Integrator 시스템에서 프로세스를 실행시키기 위해, 프로세스 정의 언어인 PML (Process Management Language)로 표현된다. 프로세스 관리 시스템의 본질 중에서, 모델의 개발은 PML과 그것을 지원하는 시스템의 속성에서 비롯된다. User model은 run-time concurrency의 요구, 데이터의 범주, UI의 요구사항, 앞으로 일어나 변화의 범주 등의 기준을 반영하여, Application Model로 변환된다. 그 모델은 다음과 같다.



[그림 10.6 : The Application Model 1]

여기서 사용된 RAD는 PML에서 프로세스를 기술하기 위한 기초이며, 시스템 role은 기본적인 언어의 상속개념을 사용한 세 가지 overlay로 구성되어 있다. *All* task role의 필수적인 행위에 대한 PML은 *WPRole*로 정의되며, 이것은 pre-defined PML role class *Role*의 subclass이다.

```
WPRole isa Role with
classes
assocs
        doneGP    :giveport DonEnt
resources
        wpEnt     : WPEnt
        thisRole:String
        whoiam    : String := ''
        doneEnt   : DonEnt
```

```
        kill     : Bool := false
        schedFormat : collof UIField:=collof UIField(
               UIField(name='startDate', label='Schedule Start'),
               UIField(name='duration', label='Duration'),
               UIText(name='requirements', label='Requirements'),
               UIField(name='notes', label='Note'))
        seeMessage      : Bool := false
        onHold          : Bool := false
        messageString   : String :='New Scheduling Information '
actions
        a_scheduleDates:{
                UserAction(agendaLabel='Task Information');
                ModifyNow
                       (agendaLabel='Task Information',
                       object = wpEnt, readOnly=true,
                       format=schedFormat)
                        }
        displayMessage:{
              ViewResourceNow
              (agendaLabel='  Message', object=messageString,
              label=whoiam);
              seeMessage:=false
                       }
        when seeMessage
        sendIt:{
                Assign(from=whoiam, to=doneEnt.task);
                Give(interaction=doneGP, gram = doneEnt);
                kill:=true
                }
      when nonnil doneEnt
termconds  kill
end with
```

[그림 10.7 : WPRole의 예]

6) The Organization Model

이것은 하나의 manager가 design engineer나 QA engineer로 구성된 software engineering staff 그룹에 대한 전체적인 책임을 맡고 있음을 보여주는 계층도이다. 그것은 프로세스가 사용자 한 그룹에 해당되기 때문에 PM은 프로세스 진행상황이나 프로세스 진화에 책임이 있음을 보여준다.

7) The Product Model

프로세스 예제는 버전 관리 하에서, 아직 확인되지 않은 모듈이나 design, object code, source code test plans, unit test packages 등으로 구성된 unit을 고려한다. 이 관리는 library service로 모델링 된다.

2.11 프로세스 모델링과 PSEE의 특징

2장에서 10장까지 설명된 프로세스 모델링과 PSEE에 대한 특징은 다음과 같다.

1) EPOS

- Planning improvement (case based planning, learning mechanisms).
- 현재 작업 모델링 지원기능이 이전에 계획된 순서들을 매우 강조한다.

2) E3

- temporal logic 표현을 통해 객체지향 구조(OO structure)를 향상시킨다.
- 클래스의 재사용을 지원한다.

3) MERLINE

- 트랜잭션(transaction), 버젼관리, 그리고 형상관리를 제공한다.
- 모델 엔지니어링 환경을 제공한다.

4) PWI

- 역할(role)을 중심으로 한 동시성을 지원한다.
- 많은 상호작용 형태들(interaction types)을 제공한다.
- 예외사항 핸들링(Exception handling)을 제공한다.
- 코드로부터 추상화 행위(abstracting behavior)를 제공한다.
- 사용자 인터페이스(User Interface) 개선 (sensible navigation among work items).

(5) SPADE

- 역할(role)과 관점(view)의 가시적인 표현(Explicit representations) 제공한다.
- Prescriptive vs. goal-oriented modeling을 제공한다.

(6) SOCCA

- Consistency checkers를 제공한다.
- 실행 가능한 형태로의 트랜잭션(translation)을 제공한다.

(7) OIKOS

- 모델링이 워크스페이스 수준에서 너무 자세하게 이루어진다.
- 보다 강력한 제품모델이 필요하다.
- 트랜젝션(transaction)들을 직접 손으로 코딩해야 한다.

● 상대적인 속성들을 표현하기 위한 로직(logic)의 개발을 지원한다.
● 프로세스 엔진의 성능과 사용자 인터페이스의 친숙함을 제공한다.

(8) ALF

● 행위 모델링(behavior modelling)을 위한 개념의 직교성(orthogonality)을 제공한다.
● Formal semantics를 제공한다.
● Planning improvement
● OMS를 위한 객체지향 개념을 제공한다.

3. 결론

지금까지 90년대 중반까지 이루어진 소프트웨어 프로세스 모델링과 PSEE에 대한 9가지의 연구를 살펴보았다. 각 연구 모두 개개의 특징을 가지고 있으나 목적은 실제 프로세스의 기술(description)과 자동화에 있다. 기존의 formality에 기초한 프로세스 모델링 연구와는 달리 90년대 말에서 지금까지의 연구는 소프트웨어 개발자들에게 익숙한 UML[9,10,11]을 이용하여 프로세스 모델링을 제공함으로서 사용자의 친숙함을 증진시키려고 노력하고 있다. 반면 지금까지의 연구와는 대조적으로 프로세스 모델링과 PSEE는 산업체에서 활성화되지 못하고 있으나, 프로젝트 관리와 더불어 업체 내의 프로세스 생성, 수행 및 관리를 지원하기 위한 도구로서는 가장 이상적이라고 볼 수 있다. 따라서 현존하는 프로세스 모델링 상의 많은 어려움들을 극복하고 PSEE 구현 및 관리의 효율성을 제시할 수 있는 새로운 형태의 PML과 PSEE의 개발을 위한 연구가 꾸준히 지속될 것으로 예상된다.

참고문헌

[1] L.Osterweil, Software process is process, too, Proceedings of the Ninth International Conference on Software Engineering, Washington, DC, 1987, 2-13

[2] Alan M. Christie, A Practical Guide to the Technology and Adoption of software Process Automation, Technical Report, CMU/SEI-94-TR-007, March, 1994

[3] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, Software Process Modelling and Technology, RESEARCH STUDIES PRESS LTD., JOHN WILEY & SONS INC.

[4] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: an environment for Software Process Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, Software Process Modelling and Technology. Research Studies Press Limited (J. Wiley), 1994.

[5] Vincenzo Ambriola , Reidar Conradi , Alfonso Fuggetta, Assessing process centered software engineering environments, ACM Transactions on Software Engineering and Methodology (TOSEM), v.6 n.3, p.283-328, July 1997

[6] S. Bandinelli, E. Di Nitto, and A. Fuggetta. Supporting cooperation in software development. Technical Report 32-95, Politecnico di Milano, 1995. Submitted for publication.

[7] Bandinelli, S. and A. Fuggetta, "Computational Reflection in Software Process Modeling: the SLANG Approach," Fifteenth International Conference on Software Engineering, IEEE Computer Society Press. pp. 144-154, Baltimore, MD, 1993.

[8] "Introduction to Artificial Intelligence", Charniak, E., McDermott, D., Addison-Wesley, 1985

[9] Elisabetta Di Nitto, Luigi Lavazza, Marco Shiavolni, Emma Tracanella, and Michele Trombetta, Deriving executable process descriptions from UML, International Conference on Software Engineering, 2002

[10] Dirk Jager, Ansgar Schleicher, and Bernhard Westfechtel, Using UML for Software Process Modeling, International Conference on Software Engineering, 1998

[11] Rik Eshuis and Roel Wieringa, Verification Support for Workflow Design with UML Activity Graphs, International Conference on Software Engineering, 2002