■ VEST(Virginia Embedded Systems Toolset)

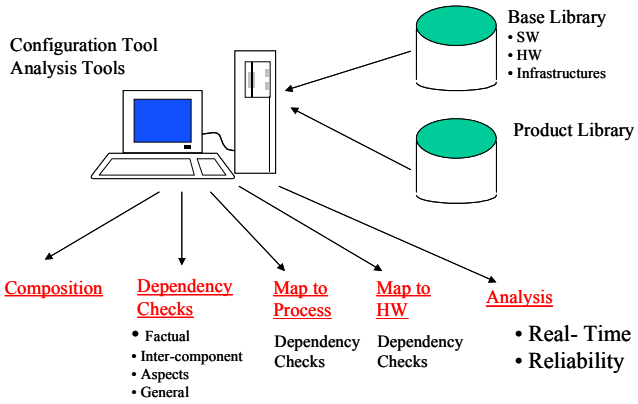
VEST[3,4,5,6]는 미국 버지니아 대학의 John A. Stankovic 교수팀이 컴포넌트기반 내장형 실시간 시스템의 개발, 구현, 평가를 개선할 목적으로 개발하였으며, 조립과 분석 아키텍처를 포함한다. VEST는 기본적인 라이브러리와 컴포넌트를 제공하며 다양한 예제의 의존성 검사를 수행하는 버전 0이 구현되었으며, 버전 1이 현재 구현 중이다.

1) 특징

VEST의 특징은 다음과 같다[4]. 첫째, 컴포넌트 기반, 실시간, 내장형 시스템을 개발, 분석하기 위한 통합 환경이다. 둘째, 수동성 컴포넌트(code fragment, function, HW등)를 생성 또는 선택한다. 셋째, 컴포넌트를 실행시간 구조(runtime structure)나 HW에 매핑(mapping)한다. 넷째, 의존성 검사(Dependency check), 비기능적 분석(non-functional analysis)을 수행한다.

2) 시스템 구조

VEST는 SW, HW 컴포넌트를 포함하는 다양한 라이브러리(Library)와 구성 도구(Configuration tool), 분석 도구(Analysis tool)를 포함하는 대화식 개발 도구로 구성된다. [그림 56]는 VEST의 아키텍처를 보여주고 있다. 라이브러리(Library)는 특정 도메인 컴포넌트를 저장하는 Base Library와 사용자 결과물을 저장하는 Product Library로 구성되며, 구성 도구는 컴포넌트 조립이나 의존성 검사를 수행하거나 컴포넌트를 Process나 HW에 매핑한다. 마지막으로, 분석 도구는 실시간, 신뢰성, 스케줄링 분석을 위해 분석 도구를 호출한다.



[그림 56] VEST의 시스템 아키텍처

가) 라이브러리(Library)

라이브러리는 특정 도메인 컴포넌트를 포함하는 base library와 사용자의 결과물(product)을 저장하기 위한 작업 디렉토리인 product library로 구성되며, 계층적인 디렉토리 구조를 갖는다. 예를 들어, OSKit component library는 Device driver, Basic C function같은 서브디렉토리들을 포함하는 디렉토리로 구성되며, [그림 57]는 VEST의 라이브러리 구조를 보여준다.

[그림 57] VEST의 라이브러리 구조

라이브러리에 저장되는 SW/HW 컴포넌트는 디렉토리에 개별 파일로서 저장되며, 계층적으로 생성 가능하다. VEST의 컴포넌트는 [표 14]와 같다.

Software Source Objects	Hardware Objects	Software Higher-Level Objects	Hardware Higher-Level Objects
Fragment/Function Component	Device	Thread Process	Composed Device System

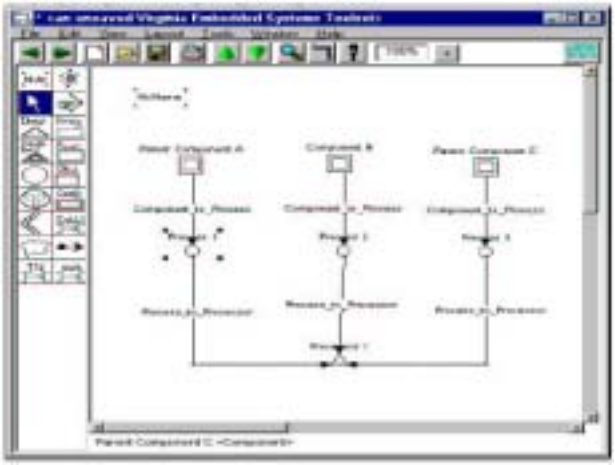
[표 14] VEST의 컴포넌트

나) 구성 도구(Configuration tool)

VEST의 구성 도구는 내장형 시스템을 구성하기 위해 새로운 컴포넌트를 생성 또는 선택하여, 시각적으로 조립하는 도구이다. 구성 도구의 기능으로는 다음과 같다. 첫째, 소프트웨어 컴포넌트를 process, thread와 같은 활성화된 실행시간 구조(active runtime structure)에 매핑(mapping)한다. 둘째, 하드웨어에 process를 바인딩(binding)한다. 셋째, 분석 도구(Analysis tool)를 호출한다. 넷째, 의존성 검사(Dependency check)를 수행한다.

3) 분석도구(Analysis tool) 호출

[그림 58]은 RM 분석(Rate-Monotonic Analysis)을 호출하는 간단한 3-process system을 보여준다. 사용자는 수동성 컴포넌트("Parent Component A", "Parent Component B", "Parent Component C")를 생성하여 실행시간 구조(runtime structure)인 Process 1, 2, 3에 매핑 한다. 그리고, 마지막으로 프로세서인 Processor 1에 세 Process를 바인딩 한다. 세 Process는 각각 2, 3, 5의 최악 수행시간(Worst-Case Execution Time: WCET)을 갖고, 7, 15, 30의 주기(period)를 갖는다고 가정하자. RM 분석 도구를 통한 스케쥴링 공식 ($\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n(2^{\frac{1}{n}} - 1)$, C: WCET, T: Period)을 통해 전체적인 효율성이 69% 이하이므로 [그림 58]의 예제는 스케쥴링이 가능하다.



[그림 58] Rate-Monotonic 분석 예제

4) 의존성 검사(Dependency checks)

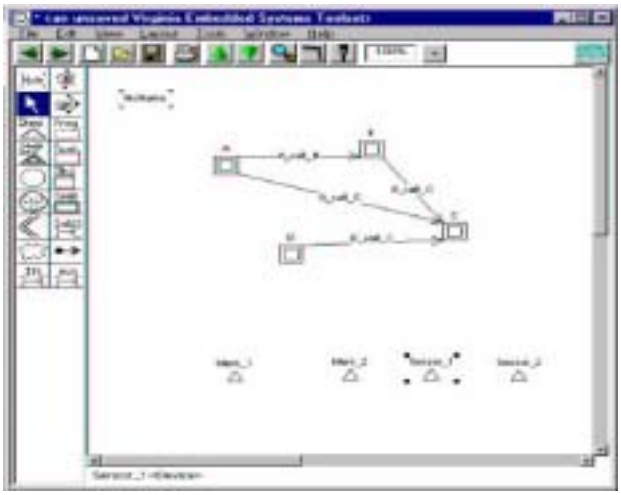
VEST의 의존성 검사는 조립된 시스템의 확신을 증가시키며, 특히 MetaH와 같은 다른 툴셋과의 큰 차이점이 된다. 의존성의 복잡성 때문에 [표 15]과 같이 4가지 형태로 나뉘며, 각 형태마다 세부적인 검사 리스트를 포함한다.

Type	Check List	Type	Check List
Factual	Worst Case Execution Time Memory needs(Memory size) Data requirements Timing constraints (Deadline, Period, Jitter requirements) Power requirements Initialization requirements	Aspect	End-to-end deadline/real-time Concurrency Persistence requirements Fault Tolerance Preemption vs non-preemption Optimizations
Inter-component	Call graphs Interface compatibility Exclusion requirements	General	No deadlock No livelock

[표 15] 의존성 검사 리스트

가) Factual Checking

Factual Checking은 가장 간단하며, 리스트는 확장 가능하다. 검사되는 의존성 요구사항 수가 많을수록 간단한 실수를 피하기 쉽다. 간단한 예로서, memory size 검사는 각 컴포넌트의 메모리 요구 사항과 하드웨어 컴포넌트에 의해 제공되는 메모리를 비교한다. [그림 59]에서 사용자는 메모리 요구사항이 각각 1, 2, 3, 4MB인 소프트웨어 컴포넌트 A, B, C, D를 배치하였다. 또한 두 개의 메모리 모듈(Mem_1, Mem_2)과 센서(Sensor_1, Sensor_2)를 배치하였다. 각 메모리 모듈은 4MB이고, Sensor_1은 2MB의 메모리를 요구한다. 결국 12MB의 메모리가 필요하나, 오직 8MB만이 제공되므로 "Not Enough Memory" 경고가 발생한다.



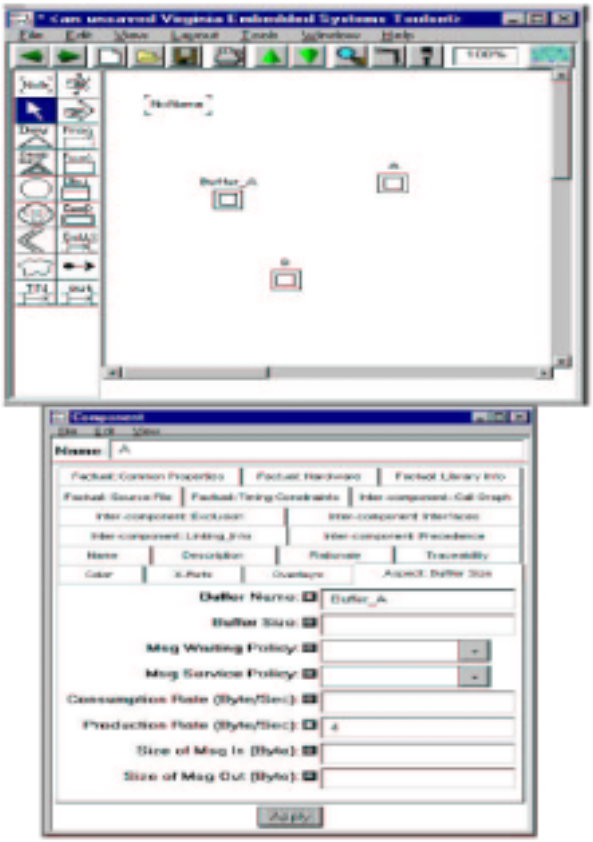
[그림 59] Memory Checking 예제

나) Inter-component

Call graphs는 호출하는 컴포넌트가 현 시스템으로부터 빠져 있는지 여부를 검사한다. interface compatibility는 반환형(return type)이 호출 컴포넌트의 반환형과 호환성(compatible)이 있는 지, 또는 입력 매개변수 리스트(input parameter list)가 호출 컴포넌트의 입력 매개변수 리스트에 호환성이 있는 지를 검사한다. exclusion requirements는 서로를 배제해야 하는 두 컴포넌트가 포함되지 않았는지를 검사한다.

다) Aspect

일반적인 절차로 명확히 캡슐화할 수 없는 문제이며, 컴포넌트의 성능(performance)에 영향을 끼친다. 예를 들면, 시스템에서 메시지 손실이 발생했는지 여부를 찾기 위해 Buffer size 문제를 검사한다. VEST는 각 컴포넌트의 메시지 손실 문제에 관련된 필요한 상호연관 정보(reflective information: 메시지 소비, 생산률, 메시지 in/out 크기)를 제공한다. [그림 60]에서 컴포넌트 A, B는 버퍼(Buffer_A)를 통하여 서로 통신하며, 컴포넌트 A가 4MB/sec로 메시지 생산하는 반면에 컴포넌트 B는 6MB/sec로 메시지를 소비한다. 메시지 손실 문제가 발생하지 않으므로 에러가 발생하지 않는다. 만약 사용자가 Buffer_A에 메시지를 보내는 또 다른 모듈 C(4MB/sec)를 추가한다면, 메시지 손실이 발생할 것이다.



[그림 60] Buffer Size 예제