번역기, 코드 생성기 및 컴파일러를 위한 검증기법 조사

김의섭, 이동아, 유준범

건국대학교 정보통신대학
서울시 광진구 화양동 1 번지 건국대학교
{atang34, ldalove, jbyoo}@konkuk.ac.kr

요 약: 시스템의 규모가 커지고 복잡해지면서 시스템을 구성하는 소프트웨어도 복잡해졌다. 이런 복잡한 소프트웨어를 설계 및 개발하기 위해서 개발자는 상위 레벨의 언어를 이용한다. 하지만 이런 상위 언어로 설계 및 구현을 할 경우 반드시 기계가 이해할 수 있는 기계어 또는 하위 언어로의 변환이 필요하다. 그리고 이런 변환이 개발자가 의도한 대로 정확히 이루어 졌는지 검증하는 과정도 필요하다. 특히 안전 필수 시스템(Safety Critical System)의 경우, 개발자의 의도대로 변환이 제대로 이루어지지 않는다면 큰 사고로 이어질 수 있기 때문에 변환에 대한 엄격하고 신뢰성 있는 검증이 필요하다. 이를 위해 본 논문은 기존 변환에 대한 검증기법들을 조사 및 분류하였다. 본 논문을 통해 개발자는 변환기의 신뢰성을 확보할 수 있는 변환 검증기법들을 확인 할 수 있다.

핵심어: 변환, 번역기, 코드 생성기, 컴파일러, 검증

1. 서 론

시스템의 규모가 커지고 복잡해짐에 따라 개발자들은 상위 레벨의 언어를 사용한 방법론을 이용하여 시스템을 개발하고 있다. 하지만 이런 상위 레벨의 언어는 개발자(사람)들이 이해 할 수 있는 언어로 구성되어 있기 때문에, 기계가 이해할 수 있는 언어(기계어, 하위 언어) 또는 하드웨어 특성에 맞는 특정한 언어로의 변환이 이루어 져야 한다. 이러한 변환을 하는 것이 변환기 즉, 번역기(Translator), 코드 생성기(Code Generator), 컴파일러(Compiler)이다.

시스템이 디지털화 되면서 소프트웨어의 수요가 늘고 있기 때문에 마찬가지로 변환기의 필요성도 늘고 있다. 안전 필수 시스템(Safety Critical System) 역시 과거 아날로그 시스템에서 디지털 시스템으로 변하고 있기 때문에 소프트웨어의 필요성이 높아지고 있고 따라서 이런 변환기를 지속적으로 필요로 하고 있다. 게다가 안전 필수 시스템에서의 오류는 큰 사고로 이어지기 때문에 변환기에 대한 엄격하고 신뢰성 있는 검증이 반드시 수행 되어야 한다. 본 논문은 이런 안전 필수 시스템에 대한 고려를 바탕으로 조사를 진행 하였고, 그 중 특히 구체적으로 다음 두 가지 사항을 염두 해 두고 진행을 하였다.

첫째, 현재 원자력 발전소의 제어 시스템은 PLC(Programmable Logic Controller) 기반의 디지털 시스템이지만, 최근 FPGA(Field-Programmable Gate Array)기반이 새로 각광 받고 있다. 시스템의 기반이 PLC 에서 FPGA 로 교체되면 시스템 개발언어가 PLC 를 구현하기 위한 C 언어 에서 FPGA 를 구현하기 위한 Verilog 로 교체된다. C 언어의 경우 이미 산업에서 신뢰성 있다고 평가된 C Compiler 를 통하여 기계어로 변환이 되기 때문에 변환에 대한 신뢰성이 문제가 되지 않지만, Verilog 의 경우에는 변환에 대한 체계적인 검증이 이루어지지 않고 있기 때문에 변환에 대한 신뢰성을 보장 할 수 없다. 따라서 이 부분이 변환기에 대한 검증이 필요하다고 여겨지는 부분이다.

둘째, 우리는 이전 연구에서 FBD(Function Block Diagram)를 Verilog 로 변환해 주는 FBDtoVerilog 프로그램과 FBD 를 C 로 변환해 주는 FBDtoC 프로그램을 개발 하였다 [1]. FBD 는 PLC 를 프로그래밍하는 언어 중 하나로써 function block 들의 연속적인 연결을 통해 PLC 의 컨트롤 행동을 시각적으로 표현하는 디자인 레벨의 언어이다. FBDtoC 와 FBDtoVerilog 프로그램은 이런 디자인 레벨의 언어인 FBD 를 C 와 Verilog 로 변환을 하는 프로그램이다. 하지만 아직 이 두 변환기 대한 체계적인 검증을 하지 못하였다. 두 프로그램 모두 안전 필수 시스템을 지원하는 도구라 엄격하고 신뢰성 있는 검증이 필요하지만 우리는 아직 검증 작업을 하지 못 하였다. 따라서 이 부분 역시 검증이 필요하다고 여겨지는 부분이다.

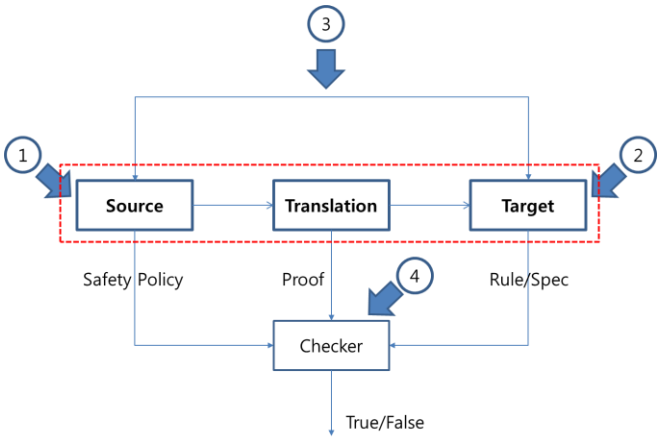
본 논문은 이 두 곳에 대한 검증이 필요하다고 생각하여 변환에 대한 검증 기법에 대한 조사를 시작하였다. 조사를 바탕으로 과거 검증기법이 중점적으로 삼고 있는 대상에 맞추어 4 가지로 관점으로 분류하였고, 각각의 관점이 충족하는 속성에 맞추어 3 가지로 분류 하였다. 특히, 각각의 관점들 중 FBDtoC 프로그램과 FBDtoVerilog 프로그램에 적용하기 적절한 기법들에 대해 가능성도 가늠해 보겠다.

본 논문은 다음과 같이 구성 된다. 2 장에서는 과거

변환에 대한 검증기법을 일정한 기준을 가지고 분류하였다. 3 장에서는 2 장에서 분류한 기준을 중심으로 몇 가지 구체적인 기법을 소개한다. 4 장에서는 2 장에서 분류한 기법 이외의 검증기법들에 대해서 소개한다. 5 장에서 결론 및 향후 연구 계획에 대해 정리한다.

2. 변환 검증기법 분류

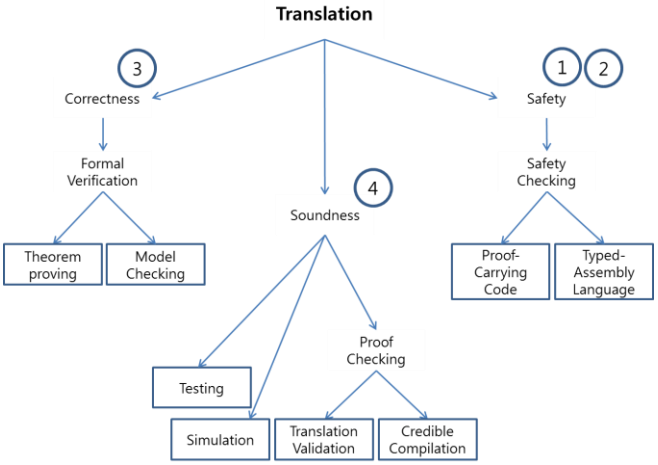
변환을 검증하는 기법은 소프트웨어의 역사만큼 오래 되었고 많이 존재한다. 다양한 기법들이 제시되었고, 각각 검증을 하려는 대상과 관점 또한 다양하다. 변환은 소스 프로그램(코드, 모델, 언어 등)이 변환기(번역기, 코드 생성기, 컴파일러 등)의 입력으로 들어와 변환 후 목적 프로그램(코드, 실행파일 등)으로 출력 되는 과정이다. 이러한 변환에 대해 검증하는 기법들을 주된 대상에 따라 4 가지 관점으로 분류 할 수 있다[그림 1]. [그림 1]의 점선 박스는 변환을 하기 위한 필수 요소 3 가지(소스코드, 목적코드, 변환기)와 변환을 나타내고 있고, ① ~ ④가 변환을 검증하는 대상에 따라 4 가지로 분류된 관점이다.



[그림 1] 검증 대상에 따른 기법들의 4 가지 분류

① 소스 프로그램: 소스 프로그램이 safety 속성을 지켜 작성 되었고 변환 되었는지를 중심으로 검증을 하는 관점.
② 목적 프로그램: 목적 프로그램이 소스 프로그램의 요구사항을 유지하며 변환 되었고 rule 을 지키고 있는지를 중심으로 검증을 하는 관점.
③ 소스 프로그램과 목적프로그램의 관계: 소스 프로그램과 목적 프로그램의 비교, 추론을 통해 상호간의 일치성을 확인하여 검증을 하는 관점.
④ 변환과정: 변환의 각 단계마다 변환을 검증할 수 있는 proof 를 생성하고 각각의 proof 를 확인하여 소스 프로그램과 목적 프로그램의 일치성을 우회적으로 검증을 하는 관점.

[그림 1]의 ① ~ ④의 관점에 맞추어 해당 관점이 충족하는 속성을 세부적으로 3 가지(correctness, soundness, safety)로 분류 할 수 있다[그림 2].



[그림 2] 검증 대상에 따른 기법들이 충족하는 속성

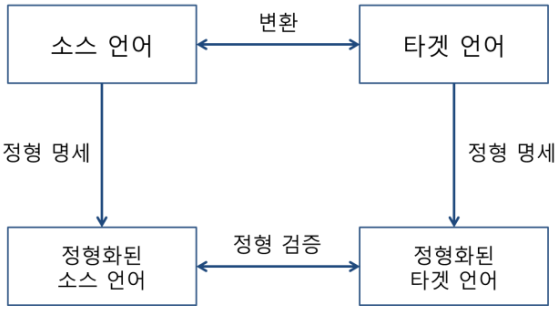
① Safety - Proof-Carrying Code(PPC)
온전하지 않는 입력이 들어오면 온전하지 않은 결과를 출력하게 된다. PPC 는 소스 프로그램이 온전하게 작성된 프로그램인지 safety police 를 이용하여 검증을 하고, 변환이 해당 safety police 를 지키며 변환을 하는지 검증.
② Safety - Typed Assembly Language(TAL)
변환된 프로그램은 해당 프로그램이 동작하는 환경에 맞게 작성 되야 한다. TAL 는 변환된 프로그램이 실행 실행 환경에 맞게 작성 되었는지 coding rule 을 통해 확인한다. 또한, 소스 프로그램의 type 과 목적 프로그램의 type 이 서로 일치하는지 type checking 을 통해 확인을 하여 변환에 대한 일치성 여부를 확인하는 기법이다.
졌다면 오류가 발생할 일이 없다는 개념이다.
③ Correctness - Theorem proving, Model checking
해당 기법은 모든 동작에 대해 정의 하고 변환에 대해 검증을 하기 때문에 correctness 에 속한다. 정확하고 정밀도 높은 검증을 수행 할 수 있기 때문에 신뢰성 있는 결과를 제공한다.
④ Soundness – Testing, Simulation, Translation validation, Credible compilation
해당 기법은 특정한 입력, 동작, proof 에 대해서만 검증을 하기 때문에 변환기에 대한 모든 검증을 보장 하지는 못하기 때문에 soundness 에 속한다.

3. 변환 기법

3.1 Correctness Property

3.1.1 Theorem proving

Theorem proving 은 소스 프로그램과 목적 프로그램의 비교를 통해 상호간의 일치성(correctness) 확인을 통해 검증을 한다. 소스 프로그램과 목적프로그램을 정형명세 하고, 정형 명세 된 소스와 목적 프로그램 사이의 관계를 정형 검증을 통해 검증을 한다[그림 3].



[그림 3] Theorem proving 검증의 일반적인 예

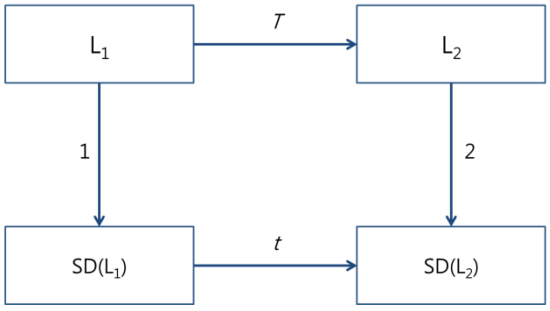
Theorem proving 은 표기 의미론(Denotational semantic)에 기초를 두고 있는 방법으로, 수학적인 심볼들을 이용하여 프로그램을 수학적인 논리로 표기하고, 논리적으로 검증을 한다. 표현방법의 대표적의 예로는 hoare's logic, first-order logic, higher-order logic 등이 있다.

검증범위는 정형 명세를 통해 모든 부분을 검증을 할 수 있다. 또한, 상호간에 일치성을 보기 때문에 신뢰성이 높은 방법이다. 하지만 사용성(시간, 노력, 지식) 측면에서 이 방법은 개발자에게 높은 수준의 전문적인 지식과 시간 및 노력을 요구한다. 따라서, 전문가가 아니면 사용하기 어렵고, 검증을 하는 과정 역시 직접 해야 하기 때문에 시간과 노력이 많이 들어간다. 경험과 노하우가 있는 전문가가 아닌 일반인이 사용하기에는 어려운 분류이다. 적용시기는 모든 변환이 이루어진 다음에 검증 과정을 진행 할 수 있다. FBDtoC 와 FBDtoVerilog 에 적용을 고려할 경우, 충분히 사용 가능하겠지만 경험과 노하우의 부족, 시간과 노력이 많이 들어가는 단점이 있어 사용하기에 용의하지 않다.

Theorem proving 을 사용한 방법으로 natural sematic 을 이용하여 Mini_ML 언어로부터 CAM 언어로 변환한 방법이 있다 [2]. Natural semantic 은 표기 의미론의 한 방법으로써 공리(Axiom)와 추론 규칙(Interface Rule)의 집합으로 구성된 표기법으로 시스템을 정형적으로 표현한다. Natural semantic 의 표기의 예는 다음과 같다

$\vdash P : \alpha$	P 는 α를 실행한다.
$\vdash P \in \pi$	P 의 타입은 π이다.
$\vdash P \rightarrow P'$	P 는 P'로 변환된다.

위와 같은 natural semantic 을 통해 dynamic semantic 과 translation 을 정의 하고, 변환의 정확성을 증명 한다.



[그림 4] Natural semantic 을 이용한 검증 개요

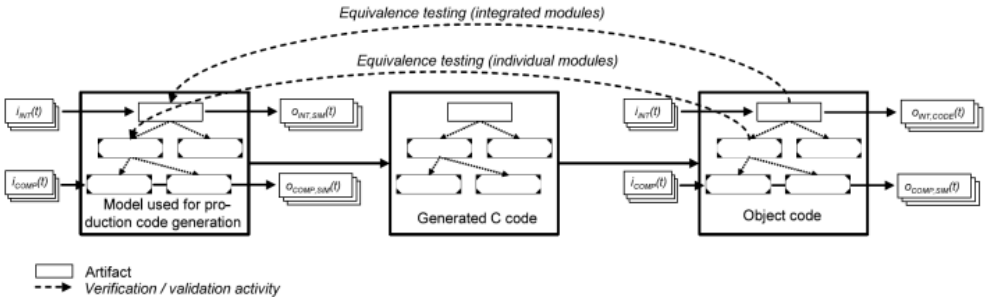
[그림 4]는 natural semantic 을 이용한 검증 개요를 나타내고 있다. 여기서 T 는 프로그램의 변환이고, t 는 semantic value 의 변환이다. L_1 의 semantic domain 은 $SD(L_1)$의 의 value 에 맵핑되고 L_2 의 semantic domain 은 $SD(L_2)$ Value 에 맵핑된다.

Theorem proving 을 이용하거나 변경한 다른 다양한 기법들이 있다. Abstract interpretation 를 이용하여 Source-to-Source 방법을 이용하여 정확도를 증명한 방법 [3], basic block 에 abstract interpretation 을 이용하여 컴파일러의 register allocation 에 위반이 있는지 검증한 방법 [4], logical framework 를 이용하여 변경이 어떻게 되었는지를 중심으로 검증한 방법 등이 있다 [5].

3.2 Soundness Property

3.2.1 Testing & Simulation

Testing & Simulation 은 소스 프로그램과 목적 프로그램의 입력(test case)과 결과를 바탕으로 상호간의 일치성을 확인하며 검증을 한다. 소스 프로그램과 목적 프로그램을 위한 입력을 작성 후, 실행을 하여 각각의 결과를 얻고, 결과의 일치성을 확인하여 검증을 하다.

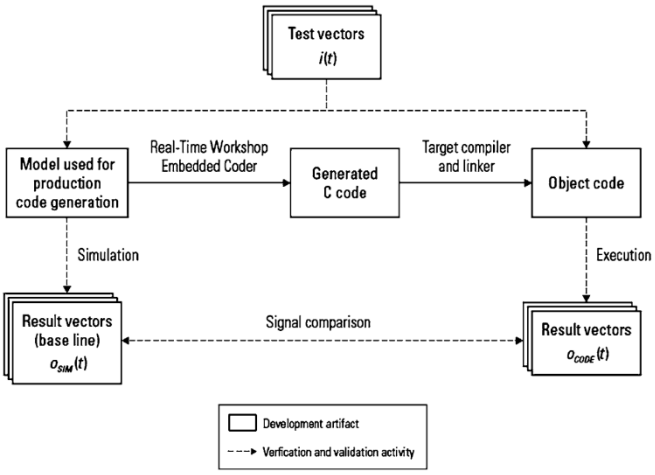


[그림 5] Individual modules 과 Integrated modules 의 Equivalence 테스팅 [6]

검증범위는 부분적으로 검증이 가능하다. 기본적으로 테스팅과 시뮬레이션은 변환기의 절대적인 정확함을 검증하는 것이 아니다. 개발자가 작성한 입력(test case)에 한에서만 변환이 온전하게(Soundness) 이루어 졌는지를 통해 검증 하는 방법이다. 따라서 절대적인 신뢰성이 필요한 시스템이라면 신중히 고려해 보야 할 필요가 있다. 사용성 측면에서 테스팅과 시뮬레이션 검증기법은 손으로 직접 수행 해야 되기 때문에 시간과 노력이 많이 들어 가지만 theorem proving 과는 달리 전문적인 지식이 많이 필요하지 않아 비 전문가가 사용하기에도 적합하다. 과거 시뮬레이션과 테스팅은 개발자가 직접 손으로 해야 했다. 따라서 시간과 노력이 많이 들었다. 하지만 현재는 테스팅과 시뮬레이션을 자동화 해주는 도구들이 많이 개발이 되어서 검증을 효율적으로 하고 싶다면 사용해 볼 하다. 적용시기는 변환기가 완성되고 변환이 모두 이루어진 다음에 검증 과정을 진행 할 수 있다. FBDtoC 와 FBDtoVerilog 에 적용할 경우, FBD 를 모델 또는 추상화된 모델이라고 보고 시뮬레이션을 하고 변환된 프로그램은 테스팅하여 서로간의 일치성을 확인하는 방법으로 충분히 사용 가능 하다. 하지만 정확한 검증을 위해 프로그램을 충분히 검증할 수 있는 적절한 입력을 생성해야 하는 단점이 있다.

특히, 현재 개발자는 코드를 바로 작성하지 않는다. 개발 생명 주기(Development Lifecycle)에 맞추어 개발을 한다. 특히 디자인 단계에서는 목적을 모델로 디자인 한 후 개발을 한다. 이런 모델을 검증하기 위해서는 과거 개발자가 손으로 직접 시뮬레이션을 해서 검증을 했었다. 하지만 기술이 발전하면서 실행 가능한 모델을 만들 수 있는 도구(ex: Simulink [7], Stateflow [8])가 개발되었고, 이것을 이용해 과거 손으로 직접 했던 시뮬레이션을 도구를 통해 자동으로 수행 할 수 있게 되었다.

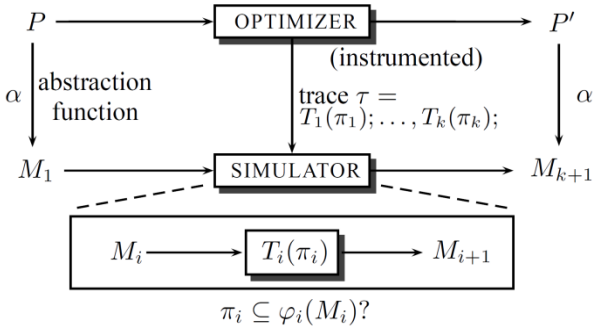
Testing & Simulation 의 한 방법으로 모델과 모델로부터 생성된 코드가 동등한지 numerical equivalence testing(comparative testing 또는 back-to-back testing)을 통하여 검증을 한 방법이 있다 [6].



[그림 6] Numerical equivalence 테스팅 [6]

[그림 5]와 [그림 6]을 통해서 numerical equivalence testing 한 방법을 이용하여 시뮬레이션과 테스팅을 하는 전체적인 개요를 확인 할 수 있다.

Numerical equivalence 테스팅 이라는 것은 시뮬레이션과 테스팅에 똑 같은 입력 값(Test vectors)을 주고 결과 값(Result vectors)을 비교하는 방법이다. 개발자가 모델링 한 대로 코드가 생성 되었다면 결과값이 같을 것이라 추론 할 수 있다. 컴파일러 최적화(Compiler Optimization)단계에 대해 시뮬레이션을 이용하여 검증을 한 방법이 있다 [9], [10]. 최적화의 scheme 는 입력과 출력의 equivalence 는 trace 를 통해 검증해 봐야 한다. [그림 7]을 통해 trace 에 대한 검증 과정의 개요를 확인할 수 있다. [그림 7]에서 τ 는 프로그램 P 에 대한 trace 를 나타내고, $T_1, \ldots, T_k$ 는 연속된 과정이다. 프로그램 P 를 추상화 한 것이 M_i 이고 $T_i(\pi_i)$가 M_i 에 대한 변환과정, M_{i+1} 이 M_i 을 시뮬레이션을 한 결과이다.



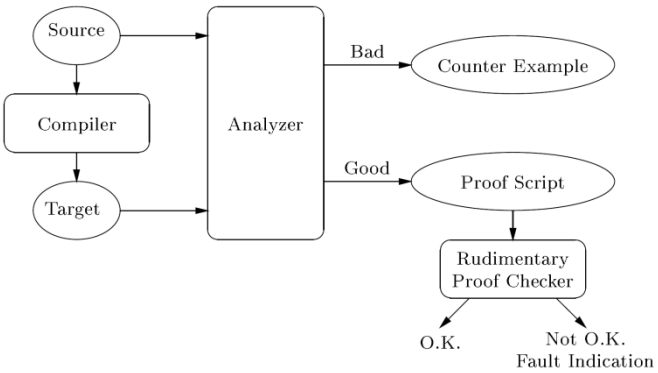
[그림 7] Trace 에 대한 검증 과정 [9]

3.2.2 Translation validation

컴파일러의 규모와 복잡성이 점점 커지고 있기 때문에 컴파일러 자체를 검증하는 일은 쉬운 일이 아니다. 이런 복잡한 컴파일러를 우회적으로 검증하기 위한 방법들이 제시 되었다. Translation validation 과 Credible compilation 은 복잡한 컴파일러를 집적적으로 검증하는 대신 우회적인 방법으로 검증을 한다.

Translation validation 는 컴파일러를 black box 로 보고 소스 언어와 목적 언어의 관계를 확인하여 일치성을 보는 방법이다. 관계의 일치성을 확인하기 위해 theorem proving, symbolic execution, simulation & testing 기법들을 사용한다.

검증범위는 부분적으로 검증이 가능하다. 변환기가 제공하는 proof 에 한에 온전하게(Soundness) 변환이 되었는지 검증을 할 수 있다. 사용성 측면에서는 변환기에 추가적으로 proof 를 생성하는 부분 작성해야 하는 점이 있어 완성된 변환기에 대한 검증을 하기 보다, 변환기를 제작하는 과정에서부터 적용해야 사용 가능하다. FBDtoC 와 FBDtoVerilog 에 적용을 고려한다면 FBDtoC 와 FBDtoVerilog 의 경우 이미 완성된 프로그램이기 때문에 적용이 용의하지 않다.

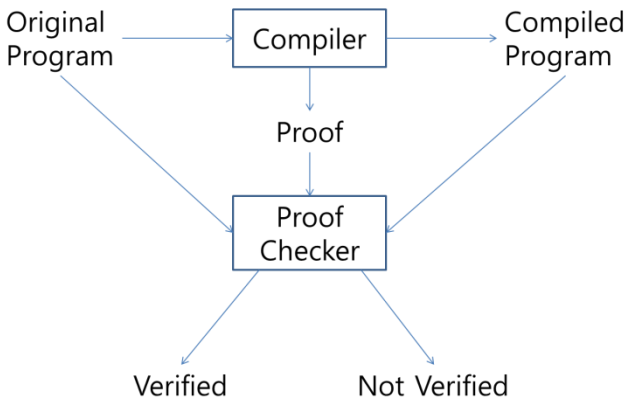


[그림 8] Translation validation 의 개요 [11]

Translation validation 이라는 개념은 1998 년에 처음 제시 됐다 [11]. Translation validation 은 컴파일러 자체를 검증하기 보다는 각각의 컴파일 단계마다 proof 를 생성해 proof 를 검사하는 것이다. [그림 8] 을 통해 전체적인 개요를 확인할 수 있다. 소스 프로그램과 목적 프로그램을 proof 를 검사하는 analyzer 의 입력으로 넣으면, analyzer 는 목적 프로그램이 소스 프로그램을 제대로 실행하고 있는지를 확인 할 수 있는 proof 통해 검사 후, 제대로 실행하고 있다면 자세한 Proof Script 를 내보내고, 그렇지 않다면 Counter Example 을 내보낸다. Counter Example 에는 목적 프로그램이 소스 프로그램과 다른 행동에 대한 시나리오가 적혀있게 된다. 따라서 Counter Example 을 통해 변환기가 수정 되어야 할 부분을 개발자에게 제공하게 된다. Proof checker 는 theorem proving, model checking, simulation 등 다양한 방법의 사용이 가능 하다.

3.2.3 Credible compilation

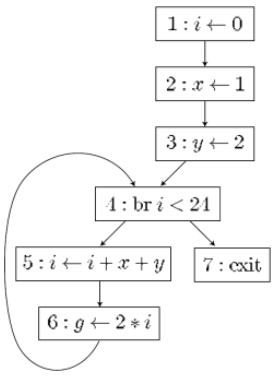
Translation validation 과 비슷하게 Proof 를 체크함으로써 검증을 하는 credible compilation 이란 기법이 있다 [12], [13]. Credible compilation 역시 컴파일 동안에 각각의 변환에 대한 Proof 를 만들어 proof 의 정당성 여부로 검증 한다. 복잡한 컴파일러를 우회적으로 검증함으로써 유용하지만, 컴파일러에 자체에 있을 수 있는 에러에 대해서는 해결 할 수 없다는 단점이 있다. 검증범위와 사용성 측면에서 translation validation 과 비슷한 성능을 발휘한다.



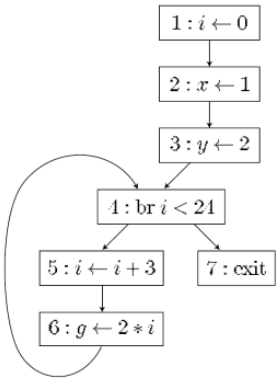
[그림 9] Credible compilation 의 개요

Credible compilation 역시 컴파일러가 모든 변환을 정확하게 한다는 것을 증명하는 대신 우회적인 방법으로 검증을 한다. 컴파일러로부터 각각의 변환에 대한 proof 를 proof checker 가 검증 함으로써 변환에

대한 일치성을 보장하는 방법이다. [그림 9]을 통해 전체적인 개요를 확인 할 수 있다.



[그림 10] 최적화 전 [12]



[그림 11] 최적화 후 [12]

컴파일 중 최적화 단계에서 컴파일러는 분석과 변환 두 단계의 활동을 한다.

1) 분석: 예제 [그림 10] 최적화 전 [그림 11] 최적화 후 을 보면, 노드 5 를 수행하기 전에 모든 부분에서 x 는 언제나 1 이고 y 는 2 이라는 분석을 할 수 있다. 따라서 노드 5 에서 x + y 는 3 이라고 분석을 할 수 있다.
2) 변환: 분석에 사용된 정보를 가지고 컴파일러는 x + y 는 3 이라는 최적화를 위한 변환을 한다.

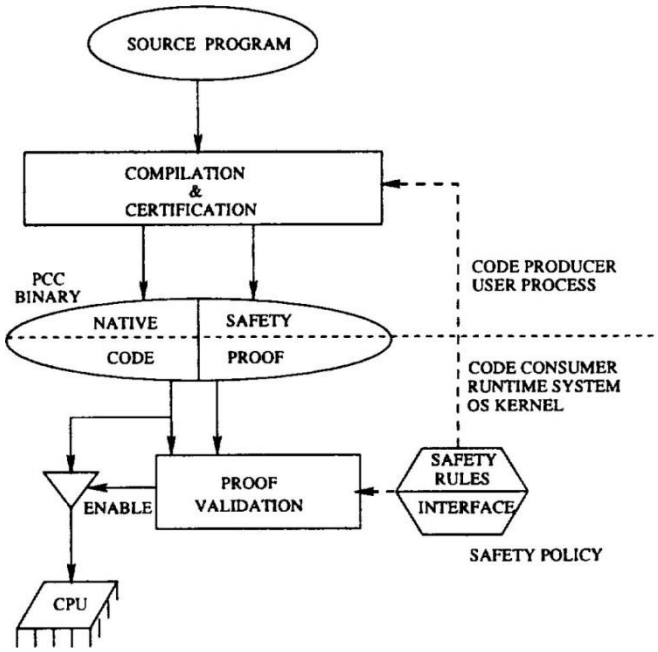
Proof checker 는 분석과 변환 두 단계가 정확히 이루어 졌는지 proof 를 생성해 정확성을 검증한다.

3.3 Safety Property

3.3.1 Proof-carrying code

Proof-carrying code(PCC)는 소스 프로그램 중심의 검증기법이다. PCC 검증기법은 소스 프로그램이 요구사항 또는 타입, 메모리 안전성 등과 같은 어떤 특정한 Spec 을 만족하는지를 safety policy 를 만들고 verified condition 을 통해 독립적으로 확인하는 검증기법이다. 특히, 신뢰성 없는 소스 프로그램을 적용해야 할 경우 유용하게 적용 가능한 기법이다. Code producer 는 native code 와 함께 safety proof 를 작성한다. Code consumer 는 safety policy 를 이용하여 작성된 safety proof 를 검사하여 안전한지 검증한다.

검증범위는 부분적으로 검증 가능하지만, safety policy 가 얼마나 강력한가에 따라 검증 강도가 결정된다. 사용성 측면에서 먼저 safety policy 를 작성해야 하는데, 이 부분은 해당 분야의 전문가에 의해 작성이 되야 한다는 점이 있다. 또한, safety proof 또한 만들어야 하기 때문에 부담이 있다. FBDtoC 와 FBDtoVerilog 에 적용할 경우 역시 safety policy 작성이 선행 되어야 하고 특히 해당 변환기는 안전필수 시스템에 적용되는 변환기이기 때문에 보다 강력한 safety policy 가 필요하다는 점이 있다.



[그림 12] PCC 의 개요 [14]

Proof carrying code(PCC)는 컴파일 된 언어가 소스 언어의 spec 과 목적 언어의 환경(H/W spec, Standard 등)에 맞는 변환을 하였는지를 검증하는 기법이다 [14], [15]. [그림 12]를 통해 전체적인 개요를 확인 할 수 있다. [그림 12]에서 safety policy 는 safety rule 과 interface 두 가지 요소로 구성 된 모습을 확인할 수 있다. PCC 는 두 요소로 이루어져 있는 safety policy 를 중심으로 이루어져 있다.

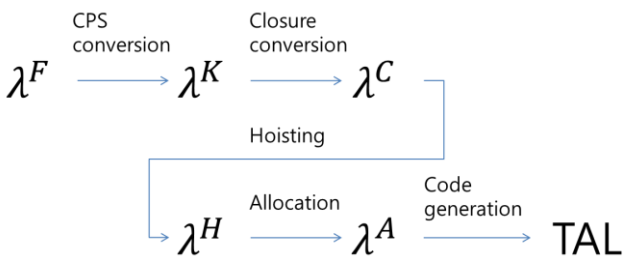
3.3.2 Typed-Assembly Language

TAL 은 소스 프로그램의 type 과 목적 프로그램의 type 을 비교하여 safety 를 보장한다. 또한 Typing rule 을 통해 Memory safety, control flow safety, type safety 등도 보장한다. TAL 과 비슷한 예로 자바 bytecode 가 있다.

적용범위는 부분적으로 검증가능하며, 사용성 측면에서는 theorem proving 과 같이 많은 정형명세와 변환 rule 을 필요로 하고 있어 용의하지만은 않다.

FBDtoC 와 FBDtoVerilog 에 적용할 경우, FBDtoC 와 FBDtoVerilog 의 목적 프로그램이 assembly language 가 아니기 때문에 적용가능 한 수준으로 맞추어 주어야 한다는 점이 있다.

TAL 을 사용한 방법으로 Strong typed assembly languages 을 사용하여 F 를 TAL 로 바꾸어 검증을 한 방법이 있다 [16]. [그림 13]은 Strong typed assembly languages 를 사용하여 F 를 TAL 로 만드는 전체적인 개요를 보여준다.



[그림 13] F 를 TAL 로 바꾸는 과정

F 언어를 TAL 로 변환을 진행 하면서 중간 프로그램인 λ^K, λ^C, λ^H, λ^A를 만들어 각 단계마다 변환을 하고 해당 단계마다의 type 을 확인해 type 이 보존되는지 확인을 통해 검증을 한다.

예를 들면 6 팩토리얼을 구하는 F 언어의 code 는 아래 첫 번째 code 와 같다. 이것을 CPS (Continuation-passing style)로 변환을 하게 되면 두 번째 code 와 같은 형태로 이루어진다. 위한 λ^F의 구문을 [그림 14] 과 같이 표현할 수 있고,, λ^K의 구문을 [그림 15] 와 같이 표현할 수 있다. λ^F에서 λ^K의 변환은 [그림 16]와 같이 표현 할 수 있다. 이를 이용해 type 을 보존하며 변환을 한다. Type 이 잘 보존된 프로그램은 오류를 발생시키지 않을 것이라는 것이 개념을 가지고 있다.

Code1

```
(fix f(n:int):int: if0(n; 1; n × f(n − 1))) 6
```

Code2

```
(fix f (n:int; k:(int) → void):
    if0(n, k(1),
        let x = n − 1 in
            f(x, λ(y:int): let z = n × y in k(z))))
(6; λ(n:int): halt[int]n)
```

types	τ, σ	$::= \alpha \mid int \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \langle\vec{\tau}\rangle$
annotated terms	e	$::= u^\tau$
terms	u	$::= x \mid i \mid \mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \langle\vec{e}\rangle \mid \pi_i(e) \mid e_1\, p\, e_2 \mid \mathsf{if0}(e_1, e_2, e_3)$
primitives	p	$::= + \mid - \mid \times$
type contexts	Δ	$::= \alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::= x_1{:}\tau_1, \ldots, x_n{:}\tau_n$

[그림 14] λ^F에 대한 구문 [16]

types	τ, σ	$::= \alpha \mid int \mid \forall[\vec{\alpha}].(\vec{\tau}) \to void \mid \langle\tau_1, \ldots, \tau_n\rangle$
annotated values	v	$::= u^\tau$
values	u	$::= x \mid i \mid \mathsf{fix}\, x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid \langle\vec{v}\rangle$
primitives	p	$::= + \mid - \mid \times$
declarations	d	$::= x = v \mid x = \pi_i\, v \mid x = v_1\, p\, v_2$
terms	e	$::= \mathsf{let}\ d\ \mathsf{in}\ e$
		$\mid\ v[\vec{\tau}](\vec{v})$
		$\mid\ \mathsf{if0}(v, e_1, e_2)$
		$\mid\ \mathsf{halt}[\tau]v$
type contexts	Δ	$::= \alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::= x_1{:}\tau_1, \ldots, x_n{:}\tau_n$

[그림 15] λ^K에 대한 구문 [16]

$$
\begin{array}{ll}
\mathcal{K}[\![\alpha]\!] & \stackrel{\text{def}}{=} \alpha \\
\mathcal{K}[\![int]\!] & \stackrel{\text{def}}{=} int \\
\mathcal{K}[\![\tau_1 \to \tau_2]\!] & \stackrel{\text{def}}{=} (\mathcal{K}[\![\tau_1]\!], \mathcal{K}_{\text{cont}}[\![\tau_2]\!]) \to void \\
\mathcal{K}[\![\forall\alpha.\tau]\!] & \stackrel{\text{def}}{=} \forall[\alpha].(\mathcal{K}_{\text{cont}}[\![\tau]\!]) \to void \\
\mathcal{K}[\![\langle\tau_1, \ldots, \tau_n\rangle]\!] & \stackrel{\text{def}}{=} \langle\mathcal{K}[\![\tau_1]\!], \ldots, \mathcal{K}[\![\tau_n]\!]\rangle \\
\mathcal{K}_{\text{cont}}[\![\tau]\!] & \stackrel{\text{def}}{=} (\mathcal{K}[\![\tau]\!]) \to void \\
\\
\mathcal{K}_{\text{prog}}[\![u^\tau]\!] & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].\mathsf{halt}[\mathcal{K}[\![\tau]\!]]x^{\mathcal{K}[\![\tau]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau]\!]} \\
\\
\mathcal{K}_{\text{exp}}[\![y^\tau]\!]k & \stackrel{\text{def}}{=} k(y^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![i^\tau]\!]k & \stackrel{\text{def}}{=} k(i^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![(\mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e)^\tau]\!]k & \stackrel{\text{def}}{=} k((\mathsf{fix}\, x(x_1{:}\mathcal{K}[\![\tau_1]\!], c{:}\mathcal{K}_{\text{cont}}[\![\tau_2]\!]).\mathcal{K}_{\text{exp}}[\![e]\!]c^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![(u_1^{\tau_1}\ u_2^{\tau_2})^\tau]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u_1^{\tau_1}]\!](\lambda x_1{:}\mathcal{K}[\![\tau_1]\!]. \\
 & \quad \mathcal{K}_{\text{exp}}[\![u_2^{\tau_2}]\!](\lambda x_2{:}\mathcal{K}[\![\tau_2]\!]. \\
 & \quad\quad x_1^{\mathcal{K}[\![\tau_1]\!]}(x_2^{\mathcal{K}[\![\tau_2]\!]}, k))^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau_1]\!]} \\
\mathcal{K}_{\text{exp}}[\![(\Lambda\alpha.u^\tau)^{\tau'}]\!]k & \stackrel{\text{def}}{=} k((\lambda[\alpha](c{:}\mathcal{K}_{\text{cont}}[\![\tau]\!]).\mathcal{K}_{\text{exp}}[\![u^\tau]\!]c^{\mathcal{K}_{\text{cont}}[\![\tau]\!]})^{\mathcal{K}[\![\tau']\!]}) \\
\mathcal{K}_{\text{exp}}[\![(u^\tau[\sigma])^{\tau'}]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].x^{\mathcal{K}[\![\tau]\!]}[\mathcal{K}[\![\sigma]\!]](k))^{\mathcal{K}_{\text{cont}}[\![\tau]\!]} \\
\mathcal{K}_{\text{exp}}[\![(\langle u_1^{\tau_1}, \ldots, u_n^{\tau_n}\rangle^\tau]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u_1^{\tau_1}]\!](\lambda x_1{:}\mathcal{K}[\![\tau_1]\!].\cdots \\
 & \quad \mathcal{K}_{\text{exp}}[\![u_n^{\tau_n}]\!](\lambda x_n{:}\mathcal{K}[\![\tau_n]\!]. \\
 & \quad\quad k(\langle x_1^{\mathcal{K}[\![\tau_1]\!]}, \ldots, x_n^{\mathcal{K}[\![\tau_n]\!]}\rangle^{\mathcal{K}[\![\tau]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau_n]\!]} \ldots)^{\mathcal{K}_{\text{cont}}[\![\tau_1]\!]} \\
\mathcal{K}_{\text{exp}}[\![\pi_i(u^\tau)^{\tau'}]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ k(y^{\mathcal{K}[\![\tau']\!]}))^{\mathcal{K}_{\text{cont}}[\![\tau]\!]} \\
\mathcal{K}_{\text{exp}}[\![e_1\, p\, e_2{}^\tau]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![e_1]\!](\lambda x_1{:}int. \\
 & \quad \mathcal{K}_{\text{exp}}[\![e_2]\!](\lambda x_2{:}int. \\
 & \quad\quad \mathsf{let}\ y = x_1\, p\, x_2\ \mathsf{in}\ k(y^{int}))^{\mathcal{K}_{\text{cont}}[\![int]\!]})^{\mathcal{K}_{\text{cont}}[\![int]\!]} \\
\mathcal{K}_{\text{exp}}[\![\mathsf{if0}(e_1, e_2, e_3)^\tau]\!]k & \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![e_1]\!](\lambda x{:}int. \\
 & \quad \mathsf{if0}(x^{int}, \mathcal{K}_{\text{exp}}[\![e_2]\!]k, \mathcal{K}_{\text{exp}}[\![e_3]\!]k))^{\mathcal{K}_{\text{cont}}[\![int]\!]}
\end{array}
$$

[그림 16] λ^F에서 λ^K의 변환 형태 [16]

4. 관련 연구

컴파일러의 경우 변환을 수행 하면서 register 나 stack, loop 등 몇 가지 점에서 error 가 나기 쉬운 지점을 가지고 있다. Register allocation 에 error 가 있는지 검증을 한 기법 [17], [18], [19]. Loop distribution, loop fusion, loop tiling, and loop interchange 등과 같은 과정에서 loop 를 다룬 기법 [20], [21], [22]. Heap 과 stack allocation 에 대해 고려함으로써 메모리 safety 를 검증하는 기법 [23], [24], [25] 등이 있다.

표 1 각 기법에 대한 비교

분류	검증대상	검증속성	사용성	검증시기	검증범위	FBDtoC & FBDtoVerilog 에 적용
Theorem Proving	소스 & 목적 프로그램	Correctness	어려움	변환 후	전체(사용자 정의)	Formal 하게 명세 후 적용 가능
Testing & Simulation	소스 & 목적 프로그램	Soundness	용의	변환 후	부분적(input)	충분한 입력 생성 후 적용 가능
Translation validation	소스 & 목적 프로그램	Soundness	보통	변환 중	부분적(proof)	Proof 를 생성하기 위한 추가적인 작업 후 적용 가능
Certifying Compiler	소스 & 목적 프로그램	Soundness	보통	변환 중	부분적(proof)	Proof 생성과 검증하는 추가적인 작업 후 적용 가능
PCC	소스 프로그램	Safety	보통	변환 전, 후	부분적(safety policy)	Safety policy 작성 후 적용 가능
TAL	목적 프로그램	Safety	보통	변환 전, 후	부분적(rule)	목적 프로그램에 대한 수정 후 적용 가능

또한, 코드 최적화 단계 역시 에러가 나기 쉬운 지점이다. 코드 최적화는 빠른 연산과 자원을 효율적으로 사용하기 위해 소스 코드를 변경하게 되기 때문에, 소스 코드의 의미를 보존하는지에 대한 검증이 필요하다. 이런 코드 최적화 단계에 있을 수 있는 오류를 다룬 검증 기법 [26], [27]들도 있다.

Model checking 을 통해 검증을 하는 기법들이 있다 [28], [29], [30], [31]. Theorem proving 과 model checking 를 같이 수행 하여 검증을 한 기법 [31]. FBD 를 verilog 와 c 로 각각 변환한 상호간의 일치성을 통해 변환기에 대해 우회적으로 검증을 한 기법 [29] 등이 있다. 이외의 추가적인 기법들 역시 과거 기법들을 조사했던 연구를 통해 확인 해 볼 수 있다 [32], [33].

5. 결론 및 향후 계획

본 논문은 변환을 검증하는 대상을 중심으로 4 가지 관점으로 분류 하였고, 해당 대상이 만족하는 속성에 따라 세부적으로 다시 3 가지로 분류 하였다. 또한, 각각의 기법에 대해 검증범위, 사용성, FBDtoC 와 FBDtoVerilog 에 적용성을 살펴 보았다. 각 기법에 대한 비교를 표 1 을 통해 확인 할 수 있다. 특히 FBDtoC 와 FBDtoVerilog 의 적용성 부분은 FBDtoC 와 FBDtoVerilog 프로그램뿐만 아니라 일반적인 변환기에 적용할 경우에 고려될 사항이라고 볼 수 있다. 각각 검증기법 마다 장단점이 있고, 특징도 다르지만 많은 기법들의 공통된 점은 아직 자동화, 상용도구, 적용 지식, 스킬 등이 부족한 상태라는 것을 확인 할 수 있었다.

현재 수많은 변환기가 존재한다. 소프트웨어의 특성이 각기 다르듯이 변환기의 특성 또한 각기 다르다. 이 수 많은 변환기들을 검증한다는 것은 굉장히 어려운 일이다. 하지만 변환에 대한 검증은 굉장히 중요한 일이기 때문에 변환기에 대한 검증은 지속적으로 필요 할 것이다.

현재 우리는 우리가 개발한 FBDtoC 와 FBDtoVerilog 를 검증할 계획에 있다. 본 논문의 결과를 바탕으로 효율적인 검증기법을 선택, 실제 적용하여 안전성을 높일 예정에 있다.

Acknowledge

본 연구는 지식경제부의 지원 및 한국원자력연구원의 안전등급제어기기 엔지니어링 도구 성능개선 기술개발사업의 연구결과로 수행되었으며(KETEP-2010-T1001-01038), 또한 2012 년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것이며(2012-0003619), 또한 한국원자력연구원 주요사업 “원자력계측제어 적합성평가, 감시 및 대응체계 구축” 사업의 지원으로 연구한 결과입니다.

참고문헌

[1] Junbeom Yoo, Jong-Hoon Lee, Sehun Jeong and Sungdeok Cha, "FBDtoVerilog: A Vendor-Independent Translation from FBDs into Verilog Programs," The Twenty-Third International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), pp.48-51, July 7-9, Miami Beach, USA, 2011.

[2] J. Despeyroux, “Proof of translation in natural semantics,” 1986.

[3] P. Cousot and R. Cousot, "Systematic Design of Program Transformation Frameworks by Abstract Interpretation," ACM SIGPLAN Notices, vol. 1, no. 37, pp. 178-190, 2002.

[4] S. M. Timothy, “Verifying the correctness of compiler Transformations on Basic Block using Abstract Interpretation,” In Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91, pp. 106-115, 1991.

[5] J. Hannan and F. Pfenning, "Compiler Verification in LF," Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh

Annual IEEE Symposium on, pp. 407-418, 1992.
[6] M. Staats and M. Heimdahl, "Partial translation verification for untrusted code-generators," Formal Methods and Software Engineering, pp. 226-237, 2008.
[7] The MathWorks, Inc: Stateflow® product page. www.mathworks.com/products/stateflow
[8] The MathWorks, Inc: Simulink® product page. www.mathworks.com/products/simulink
[9] A. Kanade, A. Sanyal and U. Khedker, "A PVS based Framework for Validating Compiler Optimizations," Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on, pp. 108-117, 2006.
[10] A. Kanade, A. Sanyal and U. Khedker, "Structuring optimizing transformations and proving them sound," Electronic Notes in Theoretical Computer Science, vol. 3, no. 176, pp. 79-95, 2007.
[11] A. Pnueli, M. Siegel and E. Singerman, "Translation validation," Tools and Algorithms for the Construction and Analysis of Systems, pp. 151-166, 1998.
[12] M. Rinard, “Credible compilation,” MIT Laboratory for Computer Science, Technical Report MIT-LCS-TR-776, 1999.
[13] M. Rinard and D. Marinov, "Credible Compilation with Pointers," s, in: Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, 1999.
[14] G. C. Necula, "Proof-carrying code," Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 106-119, 1997.
[15] G. C. Necula, "Compiling with Proofs," Carnegie Mellon University, 1998.
[16] G. Morrisett, K. Crary, N. Glew and D. Walke, "Stack-based typed assembly language," Types in Compilation, pp. 28-52, 1998.
[17] Y. Huang, B. Childers and M. Soffa, "Catching and Identifying Bugs in Register Allocation," Static Analysis, pp. 281-300, 2006.
[18] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 21, no. 5, pp. 895-913, 1999.
[19] A. Ohori, "Register Allocation by Proof Transformation," Science of Computer Programming, vol. 50, no. 1, pp. 161-187, 2004.
[20] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang and Y. Hu, "Translation and Run-Time Validation of Loop Transformations," Formal Methods in System Design, vol. 27, no. 3, pp. 335-360, 2005.
[21] L. Zuck, A. Pnueli, Y. Fang and B. Goldberg, "VOC: A Translation Validator for Optimizing Compilers," j. ucs, vol. 9, no. 3, pp. 223-247, 2003.
[22] B. Goldberg, "Translation Validation of Loop Optimizations and Software Pipelining in the TVOC Framework," Static Analysis, pp. 6-21, 2011.
[23] X. Feng, Z. Shao, A. Vaynberg, S. Xiang and Z. Ni, "Modular Verification of Assembly Code with Stack-Based Control Abstractions," ACM SIGPLAN Notices, vol. 41, no. 6, pp. 401-414, 2006.
[24] L. Jia, F. Spalding, D. Walker and N. Glew, "Certifying compilation for a language with stack allocation," Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on, pp. 407-416, 2005.
[25] G. Morrisett, K. Crary, N. Glew and D. Walke, "Stack-based typed assembly language," Types in Compilation, pp. 28-52, 1998.
[26] S. Lerner, T. Millstein and C. Chambers, "Automatically Proving the Correctness of Compiler Optimizations," ACM SIGPLAN Notices, vol. 5, no. 38, pp. 220-231, 2003.
[27] D. Lacey, N. D. Jones, E. Van Wyk and C. C. Frederiksen, "Proving Correctness of Compiler Optimizations by Temporal Logic," ACM SIGPLAN Notices, vol. 37, no. 1, pp. 283-294, 2002.
[28] M. Staats and M. Heimdahl, "Partial translation verification for untrusted code-generators," Formal Methods and Software Engineering, pp. 226-237, 2008.
[29] Dong-Ah Lee, Junbeom Yoo and Jang-Soo Lee, "Equivalence Checking between Function Block Diagrams and C Programs using HW-CBMC," The 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2011), LNCS 6894, pp.397-408, Sept. 19-21, Naples, Italy, 2011
[30] S. Jorges, T. Margaria and B. Steffen, "Assuring property conformance of code generators via model checking," Formal aspects of computing, vol. 23, no. 5, pp. 589-606, 2011.
[31] S. Owre, S. Rajan, J. Rushby, N. Shankar and M. Srivas, "PVS: Combining specification, proof checking, and model checking," Computer Aided Verification, pp. 411-414, 1996.
[32] Unknown Authors, “Compiler Optimization Verification and Maintenance”.
[33] M. A. Dave, "Compiler Verification - A Bibliography," ACM SIGSOFT Software Engineering Notes, vol. 28, no. 6, pp. 2-2, 2003.