중간표현을 이용한 재목적 컴파일러의 효율적인 테스트 방법

장한일°, 우균, 채흥석
부산대학교 컴퓨터공학과
{daystar°, woogyun, hschae}@pusan.ac.kr

An Efficient Approach to Testing Retargetable Compiler Using Intermediate Representation

Hanil Jang°, Gyun Woo, Heung-Seok Chae
Dept. of Computer Engineering, Pusan National University

요 약

컴파일러에 결함이 있다는 것은 곧 잘못된 코드를 생성한다는 것을 의미하므로 양질의 컴파일러 구성은 양질의 소프트웨어 생산을 위한 기본 요구조건이 된다. 임베디드 시스템이 널리 사용되면서 더욱 다양하고 복잡한 임베디드 프로세서가 개발되었고 이는 새로이 설계된 프로세서를 위한 새로운 컴파일러 개발의 필요를 야기하고 있다. 본 논문에서는 프로그램의 중간 표현을 기반으로 하는 효율적인 테스팅 방법을 제안한다. 언어의 구문 규칙을 모두 사용하는 테스트 케이스를 통해 컴파일러를 테스트하는 방법이 이미 연구되었으나, 기존의 소스 코드 수준의 방법으로는 테스트 케이스의 중복성이 존재하는 단점이 있다. 본 논문에서는 중간 표현의 구문 규칙을 이용해서 중복된 테스트 케이스를 제거하여 테스팅 효율을 증가시킬 수 있음을 기술한다. 또한 본 논문에서 제안하는 방법을 GCC의 중간 언어인 RTL에 적용한 예를 통해 설명한다.

1. 서론 및 연구 동기

최근 정보 가전 기술이 발달하고 실생활에 컴퓨팅의 필요가 증가함에 따라 임베디드 소프트웨어가 일반 생활에 널리 쓰이고 있다[1]. 인간의 생활에 직접 맞닿아 있는 임베디드 소프트웨어는 아주 높은 신뢰성을 필요로 한다. 만약 임베디드 소프트웨어의 개발에 사용하는 컴파일러에 결함이 있다면 잘못 생성된 코드로 인해 응용 프로그램의 결함이 야기되므로 컴파일러의 테스팅은 매우 중요한 일이다. 그리하여 컴파일러 테스팅 문제를 해결하는 여러 연구가 진행되었다[2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

이러한 추세에 발맞추어 임베디드 프로세서를 위한 재목적 컴파일러가 최근 많은 관심을 받고 있다. 재목적 컴파일러는 기존의 컴파일러를 재사용하여 새로운 컴파일러를 개발하는데 효율적이다. 현재 임베디드 시스템을 위해 새로 설계되는 프로세서가 많아지고 있으며 그로 인해 각 프로세서들을 위한 컴파일러 개발의 필요가 많아지고 있다. 이러한 상황에서 필요한 컴파일러를 완전히 새로 개발하는 대신 컴파일러를 재목적한다면 기존의 컴파일러를 간단히 수정하여 새로 개발된 프로세서를 위한 컴파일러를 신속히 개발할 수 있는 장점이 있다. 그리하여 빠르게 개발되는 재목적 컴파일러를 위한 효율적인 테스트 방법의 필요성이 대두되고 있다.

현재 연구되어 있는 컴파일러 테스팅을 위한 방법들은 프로그래밍 언어로 기술한 소스 코드를 기반으로 하여 소스 프로그래밍 언어의 문법(grammar)을 기반으로 테스트 케이스를 생성하고 있다. 하지만 컴파일러가 생성하는 기계 코드는 중간 코드에 직접적으로 영향을 받으므로 중간 표현을 기반으로 하는 테스팅이 재목적 컴파일러를 테스팅하는데 더 적합할 것으로 보인다. 이는 소스 코드가 충분히 테스트 되었다고 하더라도 중간 코드를 이용하여 철저한 테스팅을 수행하여야 하는데, 이는 소스 코드를 위한 테스트 케이스가 중간 표현 수준의 테스트 케이스에 비해 테스트 완료 기준을 충분히 만족하지 못할 수 있기 때문이다. 즉, 중간 언어를 고려하지 않고 테스트 케이스를 생성했을 때 소스 언어로 된 테스트 케이스가 커버하지 못하는 중간 코드가 있을 수 있다는 것이다. 또한, 소스 코드 레벨의 일부 테스트 케이스가 이미 다른 테스트 케이스로 해결 가능하기 때문에 소스 코드 레벨의 일부 테스트 케이스는 중복이 될 수 있기 때문이다.

이와 관련해서, 테스트 과정에는 적절한 테스트 데이터를 정의하고 예상되는 출력을 결정하여 테스트 케이스를 실행하고 실행 결과를 평가하는 것과 같은 많은 일이 필요하다. 그렇기 때문에 테스팅 비용을 줄이기 위해 테

스트 케이스의 크기를 최소화 하는 것이 아주 바람직하다. 이러한 점에서 중복 테스트 케이스를 제거하는 방법을 통해 테스트 케이스 생성 방법을 개선하는 것이 필요하다고 하겠다.

그리하여 본 논문에서는 현재의 컴파일러 테스팅 작업을 개선하고 재목적 컴파일러의 효율적인 테스트를 위해 프로그램의 중간 표현을 기반으로 하는 테스팅 방법을 제안한다. 컴파일러가 생성하는 최종 코드가 중간 코드에서 나오는 직접적인 출력이기 때문에, 소스 코드 보다는 중간 코드에 근거하여 테스트 케이스의 효율성과 테스트 완료 기준을 논하는 것이 바람직하다. 본 논문에서는 중간 표현 수준에서의 커버리지를 기반으로 하여 중복된 테스트 케이스를 제거해서 효율적으로 컴파일러를 테스트하는 방법을 제시한다. 본 논문에서는 중간표현 RTL을 사용하여 RTL의 관점에서 선별하는 방법에 대해 기술한다. 그리고 C 언어에 적용하여 사례 연구를 수행 하여, 기존 컴파일러 테스팅 기법을 따르면 많은 중복된 테스트 케이스가 생성될 수 있으며 본 연구에서 제안하는 중간 표현 기반의 방법을 적용하여 기존의 기법을 향상시킬 수 있음을 발견하였다.

본 논문은 다음과 같이 구성된다. 2절에서는 컴파일러 테스팅과 RTL에 대해 간략히 기술하고 3절에서는 RTL 기반의 효율적인 테스트 케이스 생성 방법을 제시하고 4절에서는 사례 연구의 결과를 기술한다. 5절에서는 관련 연구에 대해 기술하고 6절에서 결론을 맺는다.

2. 연구 배경

이 절에서는 컴파일러 테스팅 기법과 본 논문에서 중간 표현으로 고려한 RTL의 개요에 대해 간략히 기술한다.

2.1. 컴파일러 테스팅

테스트란 결함을 찾아내어 소프트웨어의 품질을 평가하고 향상시키는 활동이다[1]. 소프트웨어 테스팅은 무한의 테스트 케이스의 집합에서 적절하게 선별한 테스트 케이스의 유한 집합으로 대상 프로그램을 구동하여 그 동작과 예상되는 동작을 대조하는 동적 검사로 이루어진다. 테스팅은 주로 테스트 케이스에 의존적으로 되기 때문에 테스트 케이스의 생성은 소프트웨어 테스팅에서 가장 중요한 것으로 여겨진다. 테스팅 방법을 테스트 케이스 생성에 사용되는 정보에 따라 분류하자면, 소프트웨어가 어떻게 설계되고 코딩되었는지에 대한 정보를 가지고 테스트를 하는 화이트박스(또는 유리박스) 테스팅 기법과 테스트 케이스가 소프트웨어의 명세만을 사용하는 블랙박스 테스팅 기법으로 나눌 수 있다. 컴파일러는 매우 규모가 크며 소스 코드에서 목적 코드로의 생성만이 고려되므로 블랙박스 테스팅이 컴파일러의 테스팅에 더욱 적당하다.

컴파일러 테스팅에서 컴파일러가 올바른 목적 코드를 생성하는지를 시험해보기 위해서는 다양한 소스 프로그램을 테스트 데이터로 사용해야 한다. 이론적으로는 철저한 테스팅을 위해서는 무한한 소스 프로그램으로 테스팅을 수행하여야 한다. 하지만 무한한 테스트 케이스를 사용하는 것은 실제로 불가능하므로 무한 집합과 동등한 결함을 찾아낼 것이라 예상되는 테스트 케이스의 유한 집합(테스트 수트(test suite))을 선별하여 사용한다. 이는 합리적인 노력으로 테스팅을 수행하기 위해 테스팅의 효율 역시 중요하게 고려되어야 하기 때문이다.

일반적으로 테스팅 커버리지의 개념은 테스트 가능성 정의와 시스템적인 방법으로 테스트 케이스를 생성하는 것을 바탕으로 생각할 수 있다. 소스 언어의 구문이 소스 프로그램의 구조적 구성을 정의하므로 구문을 기반으로 하여 테스트 케이스를 생성하게 된다. 컴파일러 테스팅을 위한 기본 커버리지인 규칙 커버리지는 테스트 수트가 구문의 모든 규칙을 사용함을 의미한다. 즉, 모든 구문 규칙마다 적어도 하나의 테스트 케이스가 존재하면 테스트 수트가 규칙 커버리지를 만족한다는 것이다. 컴파일러 테스트를 위해서는 적어도 모든 구문 규칙이 체크되는 테스트 수트를 제공해야하며 이를 컴파일러를 테스트하기 위한 최소의 기준으로 고려할 수 있다[1, 7, 12].

2.2. RTL

RTL(Register Transfer Langugage)은 GCC(GNU Compiler Collection)에서 사용하는 중간 언어이다. RTL의 문법은 LISP의 문법과 유사하여 RTL은 명령어 패턴을 표현하는데 적합한 특징이 있다. 프로그램의 RTL 표현은 GCC의 최적화 단계에서 광범위하게 변환되고 변환된 RTL의 패턴은 코드 생성 단계에서 사용되어 컴파일러 출력을 결정한다. 따라서 RTL을 이해하는 것은 완전히 다른 기계에 GCC를 재목적할 때 매우 중요한 부분이다.

RTL은 GCC의 전단부에서 직접적으로 만드는 내부 트리 표현보다는 목표 기계에 더 가깝다. 예를 들어, RTL은 레지스터와 레지스터의 이름 같이 중간 언어에서 일반적으로 다루지 않는 부분을 다루며 RTL의 레지스터는

실제 레지스터가 아닌 pseudo 레지스터로 레지스터 할당 단계에서 실제 레지스터로 변환되는 특징이 있다. 이러한 특성상 사실 RTL 목표 기계에 가까운 중간 언어로 생각할 수 있다.

RTL 프로그램은 이중 링크로 연결된 여러 개의 RTL 표현식(RTL expression; RTX)으로 구성된다. 각 RTX는 여러 RTL 객체와 기계 명령어를 기술하는 표현식 코드(RTX 코드)를 포함한다. RTL 프로그램에 기록된 기계 명령어는 명령어 패턴으로 불리며 네이티브 기계 코드는 명령어 패턴을 이용한 패턴 매칭 절차를 통해 생성된다.

3. 효율적인 테스트 케이스 생성 프레임워크

이 절에서는 중간표현에 기반을 둔 테스트 케이스 생성 기법에 대해 설명한다. 본 논문에서는 중간 표현 언어로 RTL을 선택하였다. RTL은 GCC의 중간 표현 언어로, 현재 컴파일러 시스템 중 GCC가 재목적에 많이 활용되고 있기 때문이다.

전체 구성도는 그림 1과 같다. 테스트 케이스 생성기는 C 구문에 맞는 테스트 케이스를 생성하는 모듈로, 정의된 C 구문으로부터 각 구문 규칙을 커버하는 C 프로그램을 생성한다. 이 테스트 프로그램은 구문의 모든 생성 규칙을 커버하도록 C언어의 구문 규칙에 따라 생성된다. 여기서 테스트 케이스의 일부는 RTL 구문 규칙에서 볼 때 중복된 부분이 있을 수 있다.

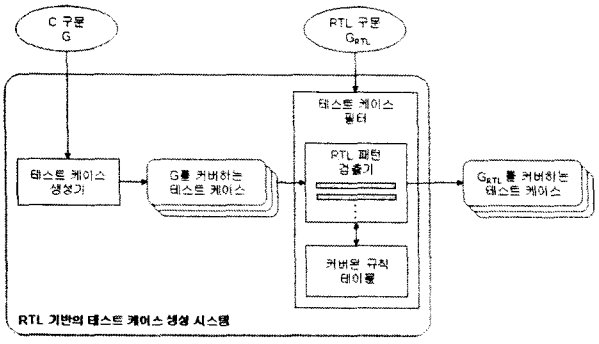

그림 1. RTL 기반의 테스트 케이스 생성을 위한 프레임워크

테스트 케이스 생성기 모듈은 RTL 구문을 고려하지 않고 테스트 케이스를 생성한다. 그런데 네이티브 코드 생성에 직접적으로 관련이 있는 것은 RTL 표현이기 때문에 앞서 언급한 바와 같이 컴파일러 후단부를 테스트하기 위한 테스트 케이스로는 RTL 구문을 커버하는 것을 위주로 쓰는 것이 테스팅 목적에 맞다. 그래서 C 프로그램 소스코드에서 RTL로 초점을 옮겨, RTL 구문 규칙을 커버하는데 도움이 되지 않는 중복된 테스트 케이스를 가려낼 수 있다.

테스트 케이스 필터는 앞서 언급한 방법에 따라 RTL의 구문 규칙을 이용해서 테스트 케이스를 선별하는 모듈이다. 이 테스트 케이스 필터는 패턴 검출기와 각 패턴 검출기가 공유하는 커버된 규칙 테이블로 구성된다. 각 패턴 검출기는 테스트 케이스로 생성된 RTX 코드를 분석하여 해당 테스트 케이스가 RTL 구문을 커버하는지 아닌지를 검사한다. 패턴 검출기가 현재 테스트 케이스로 커버되는 RTL 규칙을 찾으면 규칙 테이블을 검색하여 패턴 검출기가 찾은 RTL 규칙이 아직 사용되지 않은 규칙인지 판별한다. 아직 사용하지 않은 규칙인 경우, 테스트 케이스 필터는 이 테스트 케이스를 필터링 하지 않고 규칙 테이블에 사용된 규칙을 기록한다. 이 방법을 이용하여 중간 표현 언어를 기반으로 하는 가장 효율적인 테스트 케이스 집합을 찾을 수 있다. 이 방법은 테스트 케이스 필터에서 걸러지지 않은 모든 테스트 프로그램이 적어도 하나의 RTL 구문 규칙을 커버함을 보장한다.

4. 사례 연구

본 논문에서 제시한 방법의 효율성을 확인하기 위해 재목적 가능한 C 컴파일러로 테스트 케이스 생성기와 테스트 케이스 필터를 테스트하였다. 현재 테스트 케이스 생성기는 완전히 자동화되지는 않았다. 실제로 본 실험에서는 최초 테스트 케이스를 구문 규칙에 따라 수동으로 직접 생성하였다. 현재는 이를 단순한 작업을 통해 해결하였지만, 프로그래밍 언어의 구문 규칙을 기반으로 테스트 케이스를 자동으로 생성한 보고[4, 10]의 내용을 이용하면 자동으로 구현이 가능한 부분이다.

테스트 케이스 생성기와 테스트 케이스 필터를 구성하기 위해서는 잘 정의된 구문 규칙이 필요하다. 테스트 케이스 생성기에서는 ISO:9899의 표준 C 구문[13]을 이용하였다. 테스트 케이스 필터에서는 RTL 구문 규칙이 필요하였으나 별도로 정의된 RTL 구문 규칙을 찾을 수 없어 본 작업에서 GCC 문서[14]를 이용하여 RTX 코드를 생성하는 RTL 구문 규칙을 정의하였다.

테스트 케이스로 표준 C 구문을 커버하는 65개의 C 프로그램을 생성하였다. 표준 C 구문을 기반으로 테스트 케이스를 생성하는데 사용한 방법은 아래와 같다.

본 논문에서는 언어의 구문을 기반으로 하여 테스트 케이스를 생성하였다. 그런데 언어의 구문은 문장

(sentence), 즉 프로그램을 생성하기 위한 기초 규칙이다. 구문은 언어의 의미(semantic)를 담고 있지 않으므로 C 구문만으로는 잘 수행되는 C 프로그램을 구성할 수 없다. 그렇기 때문에 프로그램 수행을 위한 기초 문장들을 미리 정의해 두고 기초 문장에 C 구문을 하나씩 커버하는 문장을 추가하는 방법으로 테스트 케이스를 작성하였다. 특히 변수 사용에 관해서는 구문의 정의만으로는 올바른 C 프로그램을 완성할 수 없기 때문에 그런 경우에는 특정 구문을 커버하는 문장에서 사용하는 변수에 관한 선언을 별도로 추가하도록 하였다. C 구문의 규칙들을 모두 하나씩 사용하도록 하되, 프로그램 수행을 위한 기초 문장에서 이미 사용하는 규칙과 어떠한 구문 규칙을 커버하기 위해 사용한 규칙은 전체 C 구문에서 이미 사용한 구문으로 표시하여, 가능하면 하나의 구문 규칙이 하나의 테스트 케이스가 되도록 테스트 케이스를 작성하였다.

작성한 테스트 케이스를 테스트 케이스 필터로 필터링하여 RTL 구문을 중복하여 커버하는 테스트 케이스를 제거하였다. 이 작업에서 테스트 케이스를 RTL 프로그램으로 변환하는데 GCC 3.3.5버전의 i686-pc-linux 포트를 사용하였다.

	구문 규칙의 수	구문 규칙을 커버하는 테스트 케이스의 수
C 문법	193	65
RTL 문법	53	22
비율	27.5%	33.8%

표 1. RTL 기반 프레임워크의 효율

실험결과는 표 1과 같다. 실험 결과에 따르면 생성된 초기의 테스트 케이스에서 66.2%가 필터링 되었다. 이는 초기의 테스트 케이스가 커버하는 RTL 규칙을 커버하는데 33.8%의 테스트 케이스 만으로 충분한 결과를 냄을 의미한다.

이렇게 개선할 수 있게 된 이유는 구문 규칙 수의 차이에서 찾을 수 있다. 표 1을 보면 C 구문 규칙의 수가 RTL 구문 규칙의 수보다 훨씬 많음을 볼 수 있다. RTL 구문 규칙 수는 C 구문 규칙 수의 27.5% 밖에 되지 않는다. 즉, C 구문이 RTL 구문에 비해 3.6배 큼을 의미한다.

이러한 점에서 커버리지 특징을 잃지 않으면서 테스팅의 효율을 얻을 수 있었다. 게다가 본 논문의 방법은 재목적 과정에서 후단부만 수정하여 재목적 가능한 컴파일러를 얻는 경우에 더욱 합리적인 테스팅 방법이 될 수 있다.

테스트 케이스의 테스트 완료 기준을 고려할 때 테스트 케이스는 RTL 구문 규칙을 모두 커버해야 한다. 하지만 실험 과정에서 초기의 테스트 케이스가 모든 RTL 구문 규칙을 커버하지 못함을 발견하였다. 본 실험에서 커버하지 못한 RTL 구문 규칙을 실험하는 과정에서 이 문제에 대한 힌트를 얻을 수 있었다. 사실 GCC는 RTL 프로그램에 대한 광범위한 최적화를 수행하여 중간과정에서 다른 패턴으로 변형될 수 있다. 그리고 scratch, smin, rotate, abs와 같은 RTX 코드는 어떠한 C 수식에서도 생성되지 않았다. 이러한 점에서 볼 때 테스트 케이스가 테스트 완료 기준을 만족하기 위해서는 C 소스 프로그램으로 명시적으로 생성 가능한 RTL 구문 규칙의 부분집합을 정의해야 할 것으로 보인다.

5. 관련연구

컴파일러의 테스팅에 관한 여러 연구가 진행되었다. 근래에 연구된 주요 테스트 방법이나 테스트 케이스 생성 방법이 [2,3]에 조사되어 있으며 Boujarwah와 Saleh가 구문의 타입, 데이터 정의 커버리지, 문법 커버리지와 의미 커버리지 등의 비교 기준으로 컴파일러 테스팅 기술을 비교하였다[2]. 그리고 프로그래밍 언어의 문법과 의미의 형식 명세를 기반으로 컴파일러 테스트 수트를 생성하고 실행하고 수트의 질을 체크하는 방법에 대한 Kossatchev와 Posypkin의 연구[3]가 있다.

또한 프로그래밍 언어의 문법과 정적 의미의 구현물을 테스트하는 구문 기반의 방법에 대해 기술하고 구문 속성의 커버리지 기준과 커버리지 지향의 테스트 케이스 생성 전략을 제안한 연구[4, 5]가 있으며, Zelenov의 연구[8]에서는 컴파일러와 형식 텍스트 처리기를 위한 구문 변환 기반의 접근 방법이 제안되었다. Kalinov는 추상 상태 기계(Abstract State Machine, ASM)를 기반으로 하는 컴파일러 테스팅 방법을 제안하였으며[9, 10, 11], 문법 외에서도 이 연구에서 제시한 접근 방법이 Montages로 정의한 구문의 의미로 테스트 케이스 생성을 하는 것에서도 사용되었다[15]. 이 연구에서 mpC 언어의 문법과 의미를 기반으로 하는 커버리지에 관한 기준을 여러 개 제시하였으며, 여러 선행 연구에서 컴파일러 테스팅을 위한 오라클을 구현하는 문제를 해결하였다[16, 17, 18].

앞서 언급한 모든 테스팅 방법은 소스 프로그래밍 언어의 구문을 기반으로 한다. 하지만 본 논문에서는 임베디드 프로세서를 위한 재목적 컴파일러를 얻을 때 컴파일러의 후단부를 수정하게 되는 점에 초점을 두었다. 중간 표현에서 테스트 케이스를 생성하여 컴파일러의 후단부

테스트를 위한 효율적인 테스트 수트를 구성할 수 있었다.

6. 결론과 향후 과제

컴파일러의 결함은 올바르지 못한 목적 코드를 생성하므로 테스팅을 통해 컴파일러의 숨겨진 결함도 찾아낼 수 있어야 한다. 많은 컴파일러 테스팅 방법이 블랙박스 테크닉을 채택하여 소스 언어의 구문을 테스트 케이스 생성의 근거를 하고 있다. 즉, 소스 언어의 구문을 근거로 테스팅 커버리지의 기준을 정의하고 있다.

보다 효율적인 테스트 케이스 생성을 위해 본 논문에서는 중간 표현을 기반으로 하는 컴파일러 테스팅 방법을 제안하였다. 제안한 방법으로 생성된 테스트 케이스는 중간 표현 수준에서 다른 경우를 커버할 수 있는 테스트 케이스의 집합으로 이루어져있다. 제안한 방법은 전단부의 수정 없이 목적 프로세서에 따라 후단부만 수정하는 재목적 컴파일러를 테스트하는데 더 효율적으로 적용될 것이다.

앞으로 본 논문에서 제안한 방법의 자동화를 지원할 계획이다. 현재는 일부 작업을 수동으로 진행하였지만 전체 과정을 자동화하면 이 방법의 실용성을 높이는데 도움이 될 것이다. 본 논문에서 제안한 방법은 SUIF[19]의 중간 표현 같은 다른 중간 표현에도 적용되도록 확장될 수 있으며 앞으로 중간 표현을 기반으로 하는 커버리지 기준을 제안할 것이다. 본 연구 내용과 차후 연구를 통해 소스 언어가 아닌 중간 표현을 기반으로 하는 커버리지 기준을 이용하여 적절하고 효율적인 테스트 케이스 생성이 가능할 것으로 예상한다.

참고문헌

[1] Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold. (1983)

[2] Boujarwah, A., Saleh, K.: Compiler Test Case Generation Methods: A Survey and Assessment. *Information and Software Technology* 39(9) (1997) 617-625

[3] Kossatchev, A., Posypkin, M.: Survey of Compiler Testing Methods. Programming and Computer Software 31(1) (2005) 10-19

[4] Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. *Informatica 24(3)* (2000)

[5] Hannan, J., Pfenning, E.: Compiler Verification in LF. In: *Proc. of IEEE Symp. on Logic in Computer Science* (1992) 407-418

[6] Lämmel, R.: Grammar Testing. In: *Proc. of FASE* (2001) 201-216

[7] Gargantini, A., Riccobene, E.: ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *In: Proc. of Eurocast* (2001)

[8] Zelenov, S., Zelenova, S., Kossatchev, S.: Test Generation for Compilers and Other Text Processors. *Programming and Computer Software*, 29(2) (2003) 104-111

[9] Kalinov, A, et. al: Using ASM Specication for Automatic Test Suite Generation for mpC parallel Programming Language Compiler. *In: Proc. of AS* (2002) 96-106

[10] Kalinov, A, et. al: Coverage-driven Automated Compiler Test Suite Generation. *ENTCS* (2003)

[11] Kalinov, A, et. al: Using ASM Specications for Compiler Testing. *In: Proc. of ASM* (2003)

[12] Miller, K.: A Modest Proposal for Software Testing. *IEEE Software*. 18(2) (2001) 96-98

[13] ISO/IEC 9899:1999, *Programming languages — C* 408-415

[14] Stallman, R. M.: *Using and Porting the GNU Compiler Collection*. Free Software Foundation (2001)

[15] Kutter, P., Pierantonio, Al: Montages: Specications of Realistic Programming Languages. *Journal of Universal Computer Science* 3(5) (1997) 416-442

[16] Jaramilio, C, et. al: Comparison Checking: An Approach to Avoid Debugging of Optimized Code. *In: Proc. of ASM SIGSOFT* (1999) 268-284

[17] Necula, G.: Translation Validation for an Optimizing Compiler. *In: Proc. of ASM SIGPLAN* (2000) 83-95

[18] McNerney, T.: Verifying the Correctness of Compiler Transformation on Basic Blocks Using Abstract Interpretation. *In: Proc. of Symp. on Partial Evaluation and Semantics-Based Program Manipulation.* (1991) 106-115

[19] Wilson, R. P., et. al: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices* 29(12) (1994) 31-37