무결점 소프트웨어 검증 기술

서울대학교 이광근*

1. 개 요

제대로 작동할 지를 미리 검증할 수 없는 기계설계는 없다. 제대로 서있을 지를 미리 검증할 수 없는 건축설계는 없다. 인공물들이 자연세계에서 문제 없이 작동할 지를 미리 엄밀하게 분석하는 수학적 기술들은 잘 발달해 왔다. 뉴튼 역학, 미적분 방정식, 통계역학 등이 그러한 기술들일 것이다.

소프트웨어라는 인공물에 대해서는 어떠한가? 작성한 소프트웨어가 제대로 실행될 지를 미리 엄밀하게 확인해 주는 기술들은 있는가? 이 글에서는 이러한 기술들의 배경, 동향, 키워드를 비전공자도 알기 쉽도록 간략히 정리하겠습니다.

2. 소프트웨어 오류

전문가들은 모두 알고 있듯이, 컴퓨터의 화려한 기술들은 불안하고 위태롭습니다. 그 한 원인은 소프트웨어들이 가지고 있는 많은 수의 알 수 없는 "버그"(bug)들 때문입니다. 컴퓨터의 오동작을 일으키는 소프트웨어의 틀린점들은 지금 대부분의 소프트웨어에서 항상 존재하고 있습니다.

소프트웨어의 버그는 어쩔 수 없이 생기는 천재지변이 아니고 사람이 만들어내는 실수입니다. 소프트웨어 작성이 자동화 된다고 해도, 소프트웨어를 만들어주는 그 도구들도 모두 소프트웨어이고 이것 역시 사람의 손으로 만들어진 것들 뿐입니다. 사람에 의해 기획되고 사람에 의해 만들어지지 않는 소프트웨어는 없습니다. 따라서, 소프트웨어의 버그는 외부에서 스며든 "벌레" 때문이 아니고, 소프트웨어를 만드는 사람이 잘못된 소프트웨어를 작성해서 생기는 문제입니다.

소프트웨어의 버그가 특히 불안해 지는 것은, 공기중의 산소와 같이 세상의 모든 구석구석에서 우리의 일상에 없어서는 안될 존재로 스며드는 컴퓨터 때문입니다. 지금도 이미 통신/에너지/국방/금융/교통/복지/오락 등의 인프라를 운영하는 것이 컴퓨터들이기는 하지만, "지구 = 컴퓨터"라는 슬로건이 과장이 아니게 컴퓨터가 세상을 덮어가고 있습니다.

3. 오류 자동 검증

소프트웨어의 품질의 수준을 이대로 방치할 수는 없다는 압력은 당연히 커지고 있습니다. 소프트웨어의 오류로 미국경제가 지불한 비용은 2002년 595억달러[8]이고, 이 중에서 SW개발자가 적절한 도구를 사용해서 줄일 수 있는 비용은 106억달러[8]로 추정되고 있습니다(미 백악관 IT 자문위원회에서는 이미 1999년에 이 문제에 대한 연구개발에 지속적이고 효과적인 투자를 독려하였습니다[3]). 우리 경제가 미국의 1/20 정도이므로, 국내에서 소프트웨어 오류로 지불하는 비용은 아마도 년 30억달러(3조원)일 것으로 추정됩니다.

또, 무결점 소프트웨어가 시장에서 절실히 필요해 지는 이유는 따로 부연할 필요가 없을 것 입니다. 소프트웨어가 자체로 상품인 경우는 말할 것도 없고, 모든 전자기계 제품들의 경쟁력 중에서 내장된 소프트웨어의 신뢰도가 차지하는 비중은 적어도 제품 디자인만큼은 될 것입니다.

문제는 어떻게 하면 무결점 소프트웨어를 값싸게 만들어 낼 수 있는가 인데, 다행히도 그 동안 꾸준히 연구된 성과들이 그 해결책의 하나로 부상하고 있습니다.

그 연구들의 최종 목표는 소프트웨어가 오류없이 작동할지를 실행 전에 자동으로 검증하는 기술을 이룩하는 것입니다. 소프트웨어의 오류를 자동으로 찾아주는 기술입니다. 오류가 없다면 오류가 없다고 확인해 주는 기술입니다. 소프트웨어가 생각대로 (기획한대로) 구현되었는지, 그 소프트웨어가 작동하기 전에 엄밀하고 안전하게 자동으로 확인할 수 있는 기술입니다.

4. 오류 검증 기술의 발전과정

소프트웨어 오류를 자동으로 찾는 기술은 세 박자의

* 종신회원

호흡만에 한 발 전진하는 형태입니다. 첫 번째 호흡에서는 우리가 확인할 수 있는 오류의 한계를 정확하게 정의하고, 두 번째 호흡에서는 정의한 오류의 존재를 찾는 방법을 고안하며, 세 번째 호흡에서는 그 방법을 컴퓨터가 자동으로 실행할 수 있도록 소프트웨어로 만들어 냅니다.

이 때, 각 호흡마다 애매하고 허술한 구석이 없을 때에만 비로소 한 발짝의 전진이 있게 됩니다. 오류의 정의는

모호하지 않아야 하고, 고안된 방법은 프로그램이 그렇게 정의된 오류를 품고 있다면 반드시 찾아낼 수 있어야 하고, 그 방법을 구현한 소프트웨어는 그 방법 그대로 충실히 구현되어야 합니다. "반드시"와 "충실히"라는 것은 엄밀한 증명과정을 거친다는 뜻입니다.

이 세 박자 호흡을 반복하면서 확인할 수 있는 소프트웨어 오류의 정의는 기술 발전이 진행되면서 점점 정교해 집니다. 첫 세대에서는 비교적 쉬운 소프트웨어 오류들을 확인해주는 기술이 만들어지고, 두번째 세대에서는 그보다 정교한 오류들을 확인해주는 기술이 만들어지고, 세번째 세대에서는 더욱 더 정교한 오류들을 확인해주는 기술이 만들어지고... 이렇게 하면서 궁극적으로는, 소프트웨어가 생각대로 작동할지를 자동으로 확인해주는 소프트웨어 기술을 달성해가는 것입니다.

이러한 과정을 밟으며 소프트웨어 기술이 전진한 횟수는 이제 겨우 두 발짝입니다.

5. 1세대 기술: 문법 검증 기술

1970년대에 달성된 첫 발짝은, 생긴 게 잘못된 프로그램을 자동으로 찾아내는 기술입니다. 프로그램이 제대로 작동하려면 우선 프로그램의 생긴 것이 제대로 되야 합니다. 이때 오류는 프로그램의 생김새가 틀린 것이라고 정의된 경우고, 이러한 오류를 정확히 자동으로 찾아주는 기술이 달성되었습니다. 정확히 찾아준다는 뜻은, 안전하고(sound) 빠뜨림없다는(complete) 뜻입니다. 멀쩡한 프로그램을 기형이라고 진단하는 경우(빠뜨리는 경우)도 없고, 기형인 프로그램을 멀쩡하다고 하는 경우(믿을 수 없는 경우)도 없다는 뜻이지요.

이 기술을 저는 1세대 오류잡는 기술이라고 합니다. 이 기술을 문법검증기술(parsing)이라고 하고, 완전히 완성되어 오늘날의 프로그래머가 프로그램을 짜면서 늘상 아무렇지도 않게 사용할 만큼 관심밖으로 사라져 버린 성숙한 기술입니다.

그러나 이 오류잡는 기술이 겨우 1세대인 까닭은, 생긴 건 멀쩡한데 생각대로 돌아가지 않는 소프트웨어에는 속수무책인 기술이기 때문입니다. 겉모습의 오류뿐이 아닌, 속 내용(논리)의 오류 까지 자동으로 찾아주는 기술이 필요한 것입니다.

6. 2세대 기술: 타입 검증 기술

1990년대에 빛을 보기 시작한 두 번째 기술은, 타입검증(type checking)이라는 기술입니다. 제 2세대 오류잡는 기술입니다. 이때 오류의 정의는 겉모양이 틀린 프로그램만이 아니고, 속 내용이 잘못될 수 있는 경우까지를 포함하게 됩니다. 여기서 잘못된 속 내용이란, 프로그램이 실행 중에 잘못된 값이 잘못된 계산과정에 휩쓸리는 경우로 정의됩니다.

프로그램의 실행은 일종의 계산인데, 그 계산중에는 분별있는 일들이 일어납니다. 더하기에는 숫자만 들어와야 한다던지, 결혼하기라는 계산에는 남자와 여자가 한쌍으로 있어야 한다던지, 수력발전하기에는 우라늄대신에 물이 있어야 한다던지 말입니다.

이렇게 분별있게 값들이 소통되어야 하는데, 그렇지 않은 경우를 타입에 맞지 않다고 합니다. 이러한 경우가 발생하면 프로그램의 진행은 생각대로 흐르지 못하고 급기야는 갑작스럽게 멈추고 맙니다. 휘발유가 공급되어야 할 전투기의 엔진에 물이 새어들어 비행 중에 추락해 버리듯이 말입니다.

프로그램에서 타입에 맞지 않는 계산이 실행 중에 발생할 지를 미리 검증하는 방법이 바로 타입검증입니다. 이 성과는 프로그래밍 언어를 연구하는 분야에서 달성되었는데, 프로그램의 안전한 타입검증은 그 프로그래밍 언어가 제대로 디자인된 경우에만 가능합니다. (대표적인 언어가 Standard ML, OCaml, Haskell 등입니다.)

안전한 타입검증을 갖춘 프로그래밍 언어기술은 아직은 문법검증 기술만큼 모든 프로그래머들이 늘상 사용하는 기술로 널리 퍼지지는 않았습니다. 그러나 그런 타입 검증 기술은 앞으로 모든 소프트웨어 개발자가 당연한 기술이라고 생각하면서 사용하게 될 날이 멀지 않습니다.

7. 3세대 기술: 일반조건 검증 기술

2세대 오류잡는 기술로도 아직은 미흡한 실정입니다. 실행 중에 모든 값이 분별있게 착착 흘러드는 프로그램이라고 해도, 생각대로 작동하지 않을 수 있기 때문입니다.

타입에 맞는다는 것은 실은 초보적인 조건일 뿐입니다. 휘발유만이 비행기의 엔진에 흘러든다는 것이 검증

되었다고 해도, 휘발유의 폭발력이 수준미달인 경우라면 비행기가 떨어질 수 있습니다. 라면을 끓이는 계산에 항상 라면과 물과 불이 들어선다고 해도, 물이 한 방울뿐이거나 들어선 불이 포항제철의 용광로정도라면 라면의 맛이 망쳐질 수 있습니다.

타입에 맞게 컴퓨터 프로그램이 실행되더라도 의도와는 다르게 진행되는 경우, 이러한 오류를 자동으로 검증하는 기술이 필요하게 됩니다.

최근 가속이 붙기 시작한 제 3세대 오류잡는 기술은, 이렇게 확장된 오류를 검증하는 기술을 목표로 하고 있습니다. 생긴 모습도 멀쩡하고, 실행 중에 잘못된 값이 흘러들지도 않지만, 실행 중에 가져야할 정교한 조건을 만족시킬 수 없는 프로그램, 이것을 집어내는 기술입니다.

이 기술은 지난 2-30년간의 자동증명기술(theorem proving)과 프로그램 분석기술(static analysis), 그리고 논리학(mathematical logic 혹은 computational logic)의 성과 위에서 자라고 있습니다.

이 기술의 얼개는 다음과 같습니다. 프로그램이 실행중에 만족해야 하는 정밀한 속사정이 엄밀한 논리식으로 정의됩니다. 주어진 프로그램이 실행중에 그 논리식을 거짓으로 만들 수 있는 가능성이 조금이라도 있는지를 검증합니다. 있으면, 그 프로그램은 오류가 있을 수 있는 것이고, 그 가능성이 전혀 없다고 판정되면 그 프로그램은 오류가 전혀 없는 것입니다. 오류의 가능성이 조금이라도 있으면 오류가 있다고 결론내리는 경우가 생기지만(너무 보수적이지만) 안전한 것은 보장됩니다. 오류가 없다고 결론내려지면, 정말로 없다는 것은 보장됩니다. 안전하기는 하지만 완전하지는 못한 검증입니다.

8. 현재 3세대 기술의 실용성 수준

C나 C++ 프로그램에서 흔히 발생하는 오류들을 대해서는 자동으로 소스를 분석해서 오류의 위치를 찾아주는 기술이 상용화되고 있습니다.[1] 전세계적으로 2-3개 회사가 활발히 시장을 개척하고 제품을 공급하고 있습니다. 찾아주는 오류들은, 예를 들어, 메모리 접근 오류(buffer overrun)와 메모리 관리 오류(memory leak, null dereference)들입니다. 주어진 메모리 영역 밖으로 접근하는 오류(buffer overrun)들과 사용이 끝난 메모리를 재활용하지 않는 메모리 누수 오류(memory leak)를 일으키는 프로그램 소스의 위치를 자동으로 찾아줍니다.

이러한 분석기의 분석 시간과 성능은 현장의 소프트웨어 개발자나 품질관리부서에서 만족스럽게 사용할 수 있는 수준입니다. 이러한 기술은 C 프로그램의 모든 실행 과정을 예측하면서 위의 오류가 발생할 수 있는 지점들을 빠뜨림없이 집어 줍니다.

흔히 발생하는 오류이외에, 대상 소프트웨어의 세밀한 조건을 검증해 주는 도구들은 아직 완전 자동화가 불가능하고, 검증할 수 있는 대상 소프트웨어의 크기에도 제약이 있습니다. 그러나 최근 전세계의 이 분야 모든 연구팀들이 실제 시스템 소프트웨어 검증에 사용될 수 있는 수준으로 성숙한 기술들을 한 데 모아 총력을 기울이고자 하는 노력이 진행되고 있습니다[7]. 프로그래밍 언어의 타입 시스템(type system) 연구 그룹과 프로그래밍 언어 의미기반 분석 연구(semantic-based program analysis) 그룹, 소프트웨어 공학의 엄밀한 방법(formal methods)을 연구하는 그룹 등, 관련된 모든 연구진들이 활발히 참여하고 있습니다.[2]

9. 기술 키워드

- 요약 해석(abstract interpretation): 소프트웨어의 실행되는 모든 상황을 유한하면서 빠뜨림없이 요약해서 소프트웨어가 실행 중에 하는 일을 미리 예측하는 기술입니다. 이 이론[4, 5, 6]은 70년대 말에 탄생해서 현재까지 다듬어 졌고, 현재는 이 이론의 틀 속에서 거의 모든 오류 검증 기술이 디자인되고 이해될 수 있습니다. 이 이론에 기초한 실용적인 오류 검증 도구가 상용화 되고 있습니다. 이 기술을 이용해서 Linux 소스등 다양한 오픈 소스에서 지금까지 찾아지지 않았던 오류들이 찾아지고 있고, AirBus 380의 소스를 검증하는데도 성공적으로 쓰이고 있습니다[1].
- 타입 시스템(type system): 소프트웨어가 실행중에 타입에 맞게 모든 과정이 진행될 수 있는 지를 미리 확인하도록 해 주는 논리 시스템입니다. 타입 시스템은 소프트웨어를 구성하는 소스 언어 시스템에 장착되어 그 언어로 짜여진 임의의 소프트웨어를 검증해 줍니다. 이 이론의 대부분의 내용은 [9]에 정리되어 있습니다. 타입 시스템은 형식 논리에서 개발된 추론 규칙의 모습으로 표현됩니다.
- 소프트웨어 모델 검증(software model checking): 모델 검증(model checking)은 프로그램 실행중에 발생할 수 있는 모든 상태를 고스란히 일별해

1) coverity.com, polyspace.com 등

2) qpq.csl.sri.com

보면서 만족해야 하는 성질을 항상 유지하는 지를 검증하는 기술입니다. 실행 상태의 갯수가 항상 유한한 하드웨어 회로에 대해서 성공적으로 적용되었던 기술입니다. 이 기술을 소프트웨어에 적용할 때 해결해야 하는 문제는, 소프트웨어의 실행 상태는 무한히 많이 있을 수 있다는 것입니다. 이 경우, 무한히 많은 실행 상태들을 유한한 갯수로 요약해야 합니다. 유한하게 요약된 실행 상태들에 대해서 모델 검증하게 되면 정확한 결론은 만들지 못하고 애매한 결론으로 끝날 수 있습니다. 전통적인 모델검증에 대해서는 [2]에 정리되어 있습니다.

- 엄밀한 방법(formal methods): 소프트웨어를 구현하는 단계가 아니고 디자인하는 단계에서 부터 엄밀한 방법을 동원해서 검증을 하도록 하는 기술들을 말합니다. Alloy, Perfect Developer, Raise, Z, KIV, ASM등의 도구들이 개발되어 있습니다. 이런 다양한 기술들의 많은 자료가 www.omlab.ox.ac.uk/archive/formal-methods.html에 모여 있습니다.

소프트웨어의 무결점을 완전히 보장해 주는 자동 기술은 영원히 달성되지 않을 것입니다. 하지만, 조금 더 간편하고 적은 비용으로 좀 더 많은 결점들을 자동으로 검증해 주는 기술들은 계속해서 만들어 질 것이고, 그러한 기술을 달성하고 현장에 발빠르게 적용하는 그룹들이 미래의 IT 시장 경쟁에서 우위를 차지할 것은 명확합니다.

참고문헌

[1] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196-207, June 2003.

[2] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[3] President's Information Technology Advisory Committee. Information Technology Research: Investing in Our Future. www.nitrd.gov/pitac/report, February 1999.

[4] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238-252, 1977.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511-547, 1992. Also as a tech report: Ecole Polytchnique, no. LIX/RR/92/10.

[6] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83-94, 1992.

[7] Tony Hoare and Jay Misra. Vision of a Grand Challenge project, Verified software: theories, tools, experiments. vstte.inf.ethz.ch, July 2005.

[8] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Program Office Strategic Planning and Economic Analysis Group, 2002.

[9] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

이 광 근

1987 서울대학교 계산통계학 학사
1993 전산학 박사, Univ. of Illinois at Urbana-Champaign
1993~1995 Bell Labs, Murray Hill, Software Principles Research 연구원
1998~2003 과기부 창의과제 [프로그램 분석시스템 연구단] 단장
1995~2003 한국과학기술원 전산학과 조교수/부교수
2003~현재 서울대 컴퓨터공학부 부교수
E-mail : kwang@cse.snu.ac.kr