

# 스핀 모델 체커를 이용한 온라인 게임 서버의 파티 시스템 검증

김광훈, 박민규, 최윤자<sup>†</sup>

경북대학교 컴퓨터학부

e-mail : [hun172@nate.com](mailto:hun172@nate.com) [pqrk8805@hanmail.net](mailto:pqrk8805@hanmail.net) [yuchoi76@knu.ac.kr](mailto:yuchoi76@knu.ac.kr)

## Online-Game Server Party System Verification using SPIN Model Checker

Goanghun Kim, Mingyu Park, Yunja Choi<sup>†</sup>

School of Computer Science and Engineering, Kyungpook National University

### 요 약

오늘날 신뢰할 수 있는 정보통신기술(Information and Communication Technology, ICT) 시스템의 중요성은 증대되고 있으며 그에 맞춰 고위험 시스템의 검증(Verification) 작업도 점점 체계화되고 있다. 반면 여전히 일반적인 소프트웨어들은 검증 과정을 인력에 의한 테스팅과 같은 기초적인 방법에 의존하고 있다. 본 논문에서는 그 대표적인 예인 온라인 게임 서버를 대상으로, SPIN 모델 체커(SPIN model checker)를 이용한 자동화 검증방법을 적용하는 실험적인 연구를 수행한다. 연구 결과 기준의 검증 과정으로는 파악하지 못한 오류를 파악할 수 있었고, 검증 비용도 납득할 만한 수준이라는 것을 확인하였다.

### 1. 서론

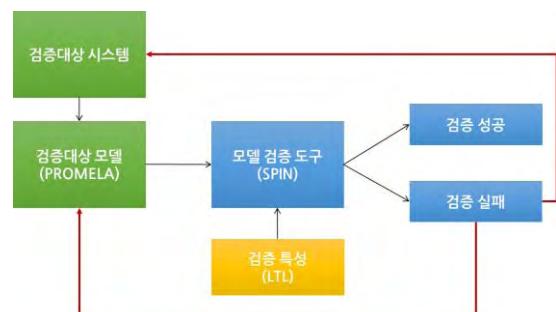
오늘날 우리는 수많은 정보통신기술(Information and Communication Technology, ICT) 시스템에 둘러싸여 있다. 가깝게는 우리 손에 들려 있는 스마트폰부터 시작해서 의료기기, 인터넷 뱅킹, 항공권 발매 시스템 등 ICT 시스템이 없는 생활은 상상하기 어렵다. 그에 맞춰 신뢰할 수 있는 ICT 시스템에 대한 요구도 커지고 있다. 신뢰할 수 없는 ICT 시스템의 경우 스마트폰 오작동과 같은 가벼운 오류를 유발할 수도 있지만 인텔 펜티엄 부동소수점 연산 오류로 인한 교환 사태 및 Therac-25 방사선 치료기기로 인한 사망사건 같은 치명적인 결과를 초래할 수도 있다<sup>[1]</sup>. 다행히 고위험 시스템(Critical system)<sup>[2]</sup>의 경우 대부분의 사람들이 시스템 신뢰도의 중요성을 잘 알고 있고 다양한 검증(Verification)<sup>[3]</sup> 방법을 사용하고 있다.

반면 오류 대응 비용이 상대적으로 작은 시스템의 경우 소프트웨어 검증 절차가 간략한 경우가 많다. 대표적으로 온라인 게임 서버를 들 수 있는데, 이는 상시 서비스를 제공하는 시스템임에도 불구하고 비용의 문제에 의해 주로 인력에 의한 테스팅과 단위 테스팅(Unit testing)에 의존하고 있다. 하지만 온라인 게임 서버는 대량의 사용자가 동시에 서비스를 제공받기 때문에, 잠재적인 오류는 다수의 사용자에게 동시에 불편을 유발할 수 있는 큰 장애를 가져올 수 있다.

따라서 본 연구에서는 온라인 게임 서버에서 특정 모듈을 사례로 엄밀한 검증방법인 모델 체킹(Model

checking)을 수행하고, 이에 대한 검증 비용이 실제로 타당한지를(feasibility) 파악하는 실험적인 연구를 수행한다. 이를 위해 온라인 게임 서버를 모델링한 예시 및 검증하고자 하는 속성을 설명하고, 실제 모델 체킹 결과 및 검증에 소요된 비용에 대해 논의한다.

### 2. 모델 체킹



(그림 1) 모델체킹 과정

모델 체킹<sup>[4]</sup>이란 시스템을 추상화한 모델에 대해, 해당 모델이 시스템에서 반드시 만족해야 할 특성을 만족하고 있는지를 확인하는 방법이다. 모델 체킹을 위해서는 우선 모델 명세 언어를 활용해 모델을 작성하고, 시제 논리를 활용해 시스템이 만족해야 하는 검증 특성을 표현해야 한다. 모델 체킹 도구는 모델과 검증 특성을 입력 받아, 수학적 논리 증명과정을 통해 해당 모델이 검증 특성을 만족하는지를 입증하고, 만약 검증이 실패한다면 이에 대한 반례를 사

용자에게 제공한다. 사용자는 이 반례를 활용해 소프트웨어에서 발생할 수 있는 위험성을 파악하고, 모델 혹은 검증 대상 시스템을 수정할 수 있다. 그림 1은 이에 대한 과정을 도식화 한 것이다.

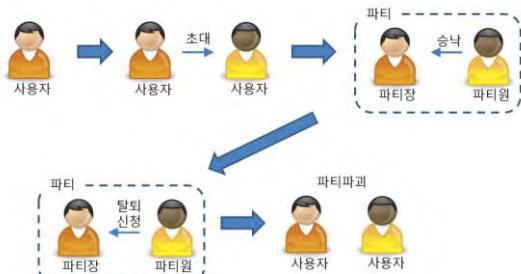
모델 검증 도구 SPIN<sup>[5,6]</sup>은 오픈 소스 소프트웨어로서 분산 소프트웨어 시스템의 검증을 위해 사용한다. SPIN에서 모델은 정형 명세 PROMELA로 표현되고, 검증 특성은 LTL(linear temporal logic)의 형태로 표현된다. PROMELA는 비결정적 오토마타 및 비동기적 분산 프로세스의 설계를 지원한다. 또한 메시지 채널을 이용해 메시지의 동기적/비동기적 통신 모델링이 가능하다. 본 논문의 검증 대상인 온라인 게임 서버의 경우, 서버-사용자 형태의 분산 시스템으로 구성되며 메시지를 통해 통신하기 때문에 해당 기능들을 제공하는 PROMELA로 표현하기 적합한 시스템이다.

SPIN은 우주 임무를 위한 다수의 알고리즘<sup>[7,8,9]</sup>이나, 차량 전장용 운영체제<sup>[10]</sup> 등의 안전중요성 시스템 검증에 활용되었다.

### 3. 온라인 게임 서버의 파티 시스템

본 논문에서 사용할 검증 대상은 현재 개발 중인 R사의 온라인 게임 T의 파티 시스템으로, 현 시스템은 아직 개발자 내부테스팅만 거친 상태이다.

게임에서의 파티 시스템은 혼자서 처리하기 어려운 적이나 임무를 여러 게임 사용자들과 협력하여 해결하게 해 주는 그룹 구성 시스템이다. 파티는 여기서의 개개 그룹을 나타내며 다수의(보통 2~5 인) 게임 사용자가 서로의 장점을 살려가며 게임을 하게 된다. 파티 내의 각 사용자를 파티원(員)이라 하고 초대, 추방 등의 권한을 가지고 있는 파티원을 파티장(長)이라 한다. 파티의 수명주기(life cycle)를 간략하게 나타내면 그림 2와 같다.



(그림 2) 온라인 게임 파티의 생명 주기

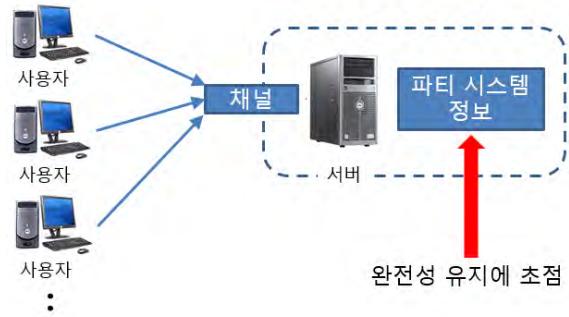
파티 시스템에서의 파티 생성/파괴, 파티원 초대/추방 등의 처리는 모두 이산적이어서 모델 체킹을 이용한 검증에 적합하다.

### 4. SPIN을 이용한 검증

#### 4.1 시스템 모델링

온라인 게임은 게임 서버 하나에 다수의 사용자가 접속하는 형태로 프로세스를 모델링할 수 있다. 또한 다수의 사용자는 서버로 동시에 메시지를 보내지만 서버에서는 하나의 메시지 큐(message queue)를 사용하므로 사용자와 서버 간 통신은 하나의 채널로 표현

했다. 그림 3은 본 시스템에 대한 개요를 나타낸다.



(그림 3) 온라인 게임 모델링

각 사용자가 서버로 보내는 메시지(mtype)는 표 1과 같으며 각 사용자들은 이 메시지를 공통의 채널을 통해 서버로 보내게 된다.

<표 1> 서버로 전송되는 메시지 종류

메시지 이름	의미
INVITE	파티원 초대
ACCEPT	초대 승낙
REFUSE	초대 거절
CANCEL	초대 취소
PROMOTE	파티장 위임
KICK	파티원 추방
LEAVE	파티 이탈

메시지, 채널, 사용자 및 파티를 모델링한 PROMELA 코드는 아래 표 2와 같다.

<표 2> 메시지, 채널, 사용자 및 파티 PROMELA 코드

```
mtype = { INVITE, ACCEPT, REFUSE, CANCEL,
          PROMOTE, KICK, LEAVE };
chan g_toServer = [1] of { mtype, byte, byte };

typedef Player{
    // 내가 현재 속한 파티, g_parties 의 인덱스이며
    // NUM_PARTIES 이면 파티에 속해있지 않음을
    // 나타낸다.
    byte partyID;
    // 현재 내가 보낸 초대.
    // 참이면 해당 사용자에게 초대를 보낸 상태이다.
    bool invitations[NUM_PLAYERS];
    // 내가 현재 보낸 전체 초대 개수
    // 동시에 일정 개수 이상의 초대는 보낼 수 없다.
    byte invitationSent;
    // 내가 받은 초대의 합
    // 동시에 일정 개수 이상의 초대는 받을 수 없다.
    byte invitationReceived;
};

Player g_players[NUM_PLAYERS];

typedef Party{
    // 파티원의 수, 0 이면 생성되지 않은 파티이다.
    byte numMembers;
    // 이 파티의 파티장, g_players 의 인덱스이다.
    byte leaderID;
};

Party g_parties[NUM_PARTIES];
```

#### 4.2 게임 사용자

게임 사용자는 악의적 사용자를 가정한다. 악의적 사용이란 정해진 프로토콜과 다르게 메시지를 보내는 것으로 해킹 시도에 해당한다. 파티 초대를 받지 않았는데도 초대 승낙 메시지를 보내는 경우가 한 예이다. 게임 사용자를 나타내는 모델은 표 3 과 같으며 해당 프로세스는 모든 메시지를 임의로 보내도록 모델링 되었다.

<표 3> 게임 사용자 프로세스 모델

```
proctype ProcPlayer(byte playerID) {
    do
        :: g_toServer ! INVITE, playerID, 0;
        :: g_toServer ! INVITE, playerID, 1;
        :: g_toServer ! INVITE, playerID, 2;
        :: g_toServer ! INVITE, playerID, 3;
        :: g_toServer ! ACCEPT, playerID, 0;
        :: g_toServer ! ACCEPT, playerID, 1;
        ...
        :: g_toServer ! KICK, playerID, 2;
        :: g_toServer ! KICK, playerID, 3;
        :: g_toServer ! LEAVE, playerID, 0;
    od
}
```

#### 4.3 게임 서버

서버는 메시지 큐에서 메시지를 하나씩 꺼내 해당 메시지 종류에 맞는 처리를 수행한다. 메시지는 메시지 종류에 해당하는 핸들러에서 처리된다.

<표 4> 서버 프로세스 모델

```
proctype ProcServer() {
    ...
    do
        :: g_toServer ? INVITE, playerID, targetID ->
            d_step { OnInvite(playerID, targetID); }
        :: g_toServer ? ACCEPT, playerID, targetID ->
            d_step { OnAccept(playerID, targetID); }
        :: g_toServer ? REFUSE, playerID, targetID ->
            d_step { OnRefuse(playerID, targetID); }
        :: g_toServer ? CANCEL, playerID, targetID ->
            d_step { OnCancel(playerID, targetID); }
        :: g_toServer ? PROMOTE, playerID, targetID ->
            d_step { OnPromote(playerID, targetID); }
        :: g_toServer ? KICK, playerID, targetID ->
            d_step { OnKick(playerID, targetID); }
        :: g_toServer ? LEAVE, playerID, targetID ->
            d_step { OnLeave(playerID); }
    od
}
```

아래 표 5 는 메시지 핸들러 중 하나인 OnInvite()를 나타낸다.

<표 5> 서버 메시지 핸들러 OnInvite()

```
inline OnInvite(playerID, targetID) {
    if
        :: ((g_parties[0].numMembers > 0) &&
            (g_parties[1].numMembers > 0))
            -> skip; // 추가 생성 가능한 파티가 없다.
        :: (playerID == targetID)
            -> printf("invalid call\n"); // 나 자신을 초대하고 있다.
        :: (g_players[targetID].partyID < NUM_PARTIES)
```

```
-> skip; // 대상이 다른 파티의 파티원이다.
::(g_players[targetID].invitationReceived
    >=MAX_INVITATION_RECEIVE)
-> skip; // 대상이 이미 최대 개수의 초대를 받았다.
:: (g_players[playerID].invitationSent
    >= MAX_INVITATION_SEND)
-> skip; // 내가 이미 최대 개수의 초대를 보냈다.
:: (g_players[playerID].invitations[targetID])
    -> printf("invalid call\n"); // 중복초대
:: ((g_players[playerID].partyID < NUM_PARTIES) &&
    (g_parties[g_players[playerID].partyID].numMembers
        >= MAX_PARTY_MEMBERS))
-> skip; // 이미 최대 파티원
:: ((g_players[playerID].partyID < NUM_PARTIES) &&
    (g_parties[g_players[playerID].partyID].leaderID
        != playerID))
-> printf("invalid call\n"); // not a leader
:: ((g_players[playerID].partyID < NUM_PARTIES) &&
    (g_players[playerID].invitationSent +
        g_parties[g_players[playerID].partyID].numMembers
        >= MAX_PARTY_MEMBERS))
-> skip; // 이미 최대 개수의 초대를 보냈다.
:: else
    ->// 초대 작업
    g_players[playerID].invitations[targetID] = true;
    g_players[playerID].invitationSent++;
    g_players[targetID].invitationReceived++;
fi
}
```

표 5 의 “:: else” 앞에 나타나는 부분들은 해당 메시지의 요청이 유효한 요청인지 검증하는 부분으로, 이는 실제 게임서버의 검증 절차와 동일하다.

사용자 및 서버 프로세스를 생성하는 초기화 모델은 아래 표 6 과 같으며, 해당 모델은 다수의 사용자 프로세스와 하나의 서버 프로세스를 생성한다.

<표 6> 초기화 모델

```
init {
    byte playerIndex;
    for (playerIndex : 0 .. (NUM_PLAYERS-1)) {
        // partyID 가 NUM_PARTIES 이면 현재 파티에
        // 속해 있지 않음을 나타낸다.
        g_players[playerIndex].partyID = NUM_PARTIES;
    }
    for (playerIndex : 0 .. (NUM_PLAYERS-1)) {
        run ProcPlayer(playerIndex);
    }
    run ProcServer();
}
```

#### 4.4 검증 특성

본 논문에서는 서버 파티 시스템 정보의 완전성(Integrity) 검증에 초점을 맞춘다. 검증할 특성을 표현한 LTL 식들은 아래 표 7 과 같으며 지면관계상 유사한 식들은 제외하였다.

<표 7> 목표 검증 특성

```
// 모든 파티는 최대 파티원 수를 초과할 수 없다.
Itl p0 { []((g_parties[0].numMembers
    <= MAX_PARTY_MEMBERS) &&
    (g_parties[1].numMembers
    <= MAX_PARTY_MEMBERS)) }
```

```

// 1인 파티는 존재하지 않는다.
ltl p1 { []((g_parties[0].numMembers != 1) &&
    (g_parties[1].numMembers != 1)) }
// 파티가 처음 생성되면 항상 파티원이 2명이다.
// (초대한 사람 + 승낙한 사람)
ltl p2 { []((g_parties[0].numMembers == 0)
    -> ((g_parties[0].numMembers == 0)W
        (g_parties[0].numMembers == 2))) }
// 파티장은 항상 해당 파티의 파티원이어야 한다.
ltl p3 { []((g_parties[0].numMembers > 0)
    -> ((g_players[g_parties[0].leaderID].partyID == 0)) }
// Party.numMembers 와 실제 파티원 수는 반드시 일치
ltl p4 { []((g_parties[1].numMembers == 0)
    -> ((g_players[0].partyID != 1) &&
        (g_players[1].partyID != 1) &&
        (g_players[2].partyID != 1) &&
        (g_players[3].partyID != 1))) }
// 파티장이 아닌 파티원은 초대를 보낼 수 없다.
ltl p5 { [](((g_players[0].partyID < NUM_PARTIES) &&
    (g_parties[g_players[0].partyID].leaderID != 0))
    -> ((g_players[0].invitationSent == 0) &&
        (!g_players[0].invitations[0]) &&
        (!g_players[0].invitations[1]) &&
        (!g_players[0].invitations[2]) &&
        (!g_players[0].invitations[3]))) }
// 파티원은 초대를 받을 수 없다.
ltl p6 { []((g_players[0].partyID < NUM_PARTIES)
    -> ((g_players[0].invitationReceived == 0) &&
        (!g_players[0].invitations[0]) &&
        (!g_players[1].invitations[0]) &&
        (!g_players[2].invitations[0]) &&
        (!g_players[3].invitations[0]))) }

```

## 5. 결과

검증을 수행한 환경은 다음과 같다.

OS : MS Windows7 64-bit

CPU : Intel i7-3770 @ 3.40GHz,

Memory : 12GB

Spin Version : 6.3.2

또한 메모리 사용량을 줄이기 위해 해시 압축(hash compaction) 옵션을 사용하였다. 아래 표 8은 각각의 특성을 검증하기 위해 소요된 자원 및 그 결과를 나타낸다.

<표 8> LTL 식 검증 결과

LTL 식	메모리(MB)	시간(sec)	오류
p0	4750.291	307	없음
p1	4750.291	302	없음
p2	7040.037	682	없음
p3	4750.291	291	없음
p4	4750.291	295	없음
p5	4750.291	296	없음
p6	399.424	0.373	있음

검증결과 LTL 식 p6의 검증이 실패했는데, 이는 파티 생성 시 파티원들의 초대 상태 초기화의 문제로 인한 것이었다. 예를 들면 초대한 사용자 U0와 승낙

한 사용자 U1으로 새로운 파티를 만드는 순간, 다른 사용자 U2가 U0와 U1에게 동시에 초대를 보낸 상태였다고 하면, 파티 생성 과정에서 U2가 U1에게 보낸 초대는 정상적으로 초기화(제거)하는데 비해 U2가 U0에게 보낸 초대는 초기화하지 않고 있었다. 반례의 분석 결과 심각한 오류는 아니었으며, 실제 시스템의 오류도 간단한 수정으로 해결할 수 있었다.

또한 검증 결과 최대 12분의 시간과, 7GB의 메모리가 소요되었으며, 이는 최근 일반적인 장비로도 충분히 소화할 수 있는 비용이다. 이와 같이 납득할 만한 비용 내에서 모델 체킹을 이용한 검증을 수행할 수 있다는 것을 확인할 수 있었다.

## 6. 결론 및 향후 연구

본 논문에서는 게임서버의 파티시스템에 SPIN 모델 체커를 이용한 자동화 검증 시스템을 적용하였다. 검증 대상이 된 파티 시스템이 어느 정도 정형화된 상태이고 내부 테스팅이 진행된 후라 심각한 오류는 발견되지 않았다. 하지만 테스팅으로 발견하지 못한 오류를 발견할 수 있었고, 상용 게임 서버 코드도 납득할 만한 비용 내에서 검증 가능하다는 것을 확인하였다.

모델 체킹은 모델링 과정에 세심한 주의가 필요하고 기존 방법에 비해 시간과 메모리 소모가 크지만 모든 상태를 검사할 수 있고 자동화되어 있다는 점에서 다중 서버 간 인증 처리나 게임 아이템 거래 같이 철저한 확인이 필요한 부분의 추가적인 검증 방법으로 충분한 가치가 있을 것으로 판단한다.

## 참고문헌

- [1] Christel Baier, et al. "Principles of Model Checking", pp.1-2, The MIT Press, 2008
- [2] Ian Sommerville, "Software Engineering 9/E", pp.291, Addison-Wesley, 2010
- [3] Christel Baier, et al. "Principles of Model Checking", pp.3-6, The MIT Press, 2008
- [4] Edmund M. Clarke, et al. "Model checking", The MIT press, 1999
- [5] GJ. Holzmann, "The model checker SPIN", IEEE Transactions on Software Engineering, Vol. 23, No. 5, 1997
- [6] Model checker SPIN - <http://spinroot.com>
- [7] Francis Schneider, et al. "Validating Requirements for Fault Tolerant Systems using Model Checking", Third IEEE Conference on Requirements Engineering, 1998
- [8] Klaus Havelund, et al. "Formal Analysis of the Remote Agent Before and After Flight", Proceedings of the 5th NASA Langley Formal Methods Workshop, 2000
- [9] GJ. Holzmann, et al. "Model-Driven Software Verification", Proc. 11th SPIN Workshop, 2004
- [10] Yunja Choi, "Model checking Trampoline OS: a case study on safety analysis for automotive software", Softw. Test., Verif. Reliab. pp.38-60, 2014