

Modeling the Active Badge System with Alloy and its Automatic Constraint Analyzer

Steven Weigao Xu

Department of Computing & Information Science

Queen's University at Kingston

Supervised by Prof. Juergen Dingel

Abstract

Formal modeling and automatic analysis were applied to the design of the Active Badge System (ABS), a system that provides a means of locating individuals within a building by determining the location of their Active Badge. This paper will discuss such a formal modeling and verification with Alloy, a new language for object modeling developed at MIT, and its constraint analyzer, a fully automatic simulation and checking tool. Key features of the ABS will be emphasized while some trivial properties of the system were ignored. It is an efficient way to characterize the conditions under which the ABS works correctly, and to find out what properties should hold or should not hold in such a system.

1. Introduction

During the past ten years, object-orientation has become very popular in software development. The most important reason for this tendency is that a number of systems can be thought of a collection of interacting objects. In supporting the specification and design of object-oriented system, object modeling always plays a very important role.

An object model is a description of the abstract state space of a system, problem domain or system environment. It shows what objects there are and how they are connected. It may describe how state changes, with multiplicity constraints and operation specs. Often, an object model can guide the structuring of a software system. Object models can be used in requirements (for describing domains and environments), in design (specifying global system state) and in coding (specifying inter-object invariants).

This paper will present how Alloy, an object modeling language, was applied to model the Active badge System.

1.1 The Active Badge System

The Active Badge system provides a means of locating individuals within a building by determining the location of their Active Badge. This small device worn by personnel transmits a unique infrared signal every few seconds. Each office, or even corridor and lobby, within a building is equipped with one or more networked sensors, which detect these transmissions. The location of the badge (and hence its wearer) can thus be determined on the basis of information provided by these sensors.

1.2 Alloy

Alloy is a new language for object modeling developed at MIT. The Alloy Constraint Analyzer is a tool for analyzing Alloy object models. The two were designed hand-in-hand: The Alloy Constraint Analyzer exploits the structure of Alloy models, and Alloy was designed to be analyzable. Alloy is aimed at the same applications as object modeling languages (such as Rational's UML).

The Alloy Constraint Analyzer is a new tool for software design that offers fully automatic analysis of object models. It can perform a deep semantic analysis of models that

incorporate complex textual constraints, rather than shallow, syntactic analysis - primarily that names are used consistently, which is offered by most current commercial tools. It can check the consistency of constraints, generate sample configurations, simulate execution of operations, and check that operations preserve constraints or an invariant. In practice, properties often don't hold, and the Alloy Constraint Analyzer can find counterexamples quickly.

2. Overview of ABS

Efficient location and coordination of staff in any large organization is a difficult and recurring problem [4]. For example, a hospital may require up-to-date information about the location of staff and patients, particularly when medical emergencies arise. A solution that provides direct location information is desirable. The Active Badge System was invented upon such a demand.

The original Active Badge System was created at Olivetti Research Laboratory (ORL) in Cambridge, UK in 1989-92 (Distributed Systems Research Group, 1999) [3]. It allows to locate people and equipments within a building determining the location of their active badges, which are small devices worn by personnel that periodically transmit infrared signals. Offices and public area in the building are equipped with sensors that detect these signals. The location of the badge (and hence its wearer) can thus be determined on the basis of information provided by these sensors.

In an active badge system, badges transmits unique infrared signal periodically (usually every 10 to 15 seconds). Each office within a building is equipped with one or more networked sensors, which detect these signals. Sensors are connected to computer network

that contains other hardware components, such as central server, that are responsible for data and queries processing. The location of the badge can be determined on the basis of information provided by these sensors. To detect active badges in transit through a building, a sensor network must provide thorough coverage through adequate placement and density of sensors. The processing result could be displayed on terminal interfaces in response to outside queries.

Active Badges can also be used as keys for access-control in high-security buildings. Some security information can be embedded into a badge so that the wearer of that badge only has access to those areas that were predefined within his/her badge. Such a function is exactly the same as that of a card-key.

In order to model the ABS, the general description about the system given above is not enough. More precise and detailed properties of the system are described as follows:

- The system employs room scale location, which means a person with a badge in this system could be located at the precision of room scale. To achieve this, infrared base sensors are installed at fixed positions (usually on the ceiling) within each room. In order to make location boundaries well-defined, based on the transmission properties of infrared signals, an assumption that signals do not pass through walls is employed here. An example of bases arrangement is shown in figure 1.

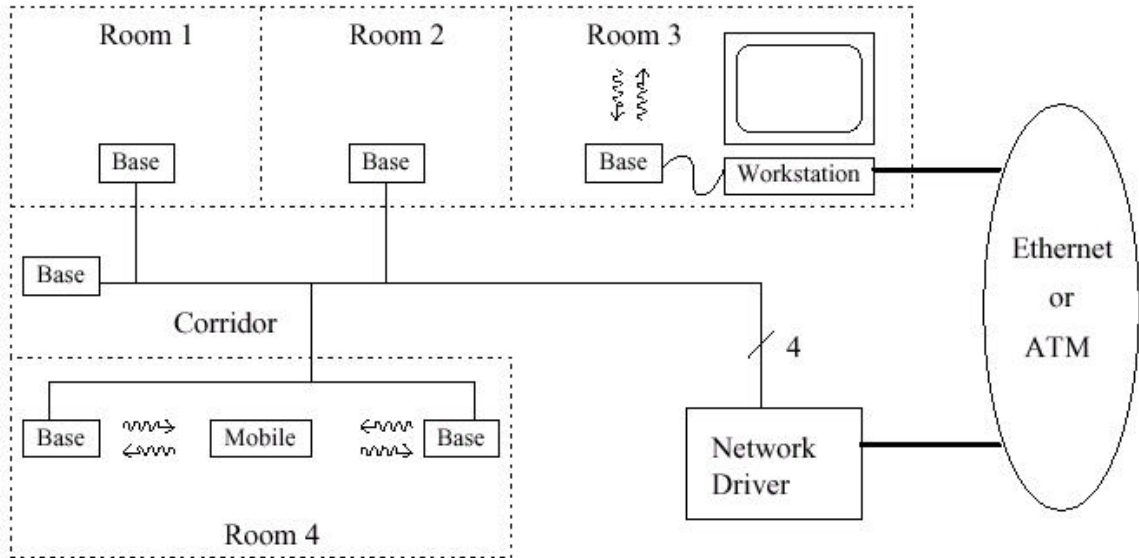


Figure 1: Infra-red sensor arrangement

- Since infrared signals are directional, and the transmission range is usually only a few meters, another assumption that signals will be detected by no more than one sensor in a large room that has more than one sensor, is practical if the granularity of sensors is well-designed.
- Each room or area in this system is associated with a level for access-control, ie. lab is assigned a level of 5 meaning that only researchers are granted rights to access, and public area is assigned level 1 meaning that everyone has access to it.
- The ABS can register, or de-register a badge if that badge enters or leaves the building via some entrances or exits. Only those bases installed at entrances or exits can register or de-register badges.
- The system provides the following operations:
 1. FIND (name): return the current location of the named badge, if any;
 2. LOOK (room): allows an investigation to be made of the badges that are currently in the specified room;

3. LOOKBASE (base): similar to LOOK (room), but the parameter is at a finer level, which is base;
4. WITH (name): find out all other badges that are currently with the named badge;
5. HISTORY (name): generate a condensed report of the location history for the named badge.

The modeling of the ABS that will be discussed below is based on the properties described above. They are basically clear enough to derive an ABS model in Alloy.

3. Overview of Alloy and its Constraint Analyzer

This paper presents how Alloy, a lightweight formal modeling notation, and the Alloy Constraint Analyzer, its automatic analyzer, were used to explore variants of the ABS' design. Before discussing the modeling of the ABS, it is essential to take a tour walking through the features and basic components of Alloy.

3.1 Features of Alloy

Alloy is a first-order notation with transitive closure that is based on predicate logic and set theories. It attempts to combine the best features of Z and UML [1]. The following distinctive features of Alloy models make Alloy distinct from other languages used in formalizing specification.

- Alloy models are declarative. A declarative model describes a system's state and behavior by listing properties or constraints [1]. Unlike a program, which is operational, it doesn't explain how states are constructed, or how execution obtains a new state from an old state. Instead, it gives constraints that define a well-formed state, and that say how a new state and an old state are related.

- Alloy models are structural. Software systems have (at least) two kinds of complexity. There is complexity due to event sequencing, and complexity in the structure of the state itself. It is the structural complexity that Alloy is designed for. There are many tools for analyzing sequences of events, but very few for analyzing structures [1].
- The Alloy models are analyzable. The language was designed hand-in-hand with a fully automatic tool that can simulate models and check their properties. Simulation generates structures and behaviors without making you provide sample inputs or test cases. Checking generates counterexamples – structures or behaviors for which an expected property does not hold; from a counterexample, it is usually not too hard to figure out what is wrong [1].
- Alloy models are micro-models. Usually orders of magnitude smaller than the systems they model. Ten or twenty lines may be enough to say something useful about a system whose implementation has thousands of lines of code. The modeling language itself is also unusually small and simple. It is easy to learn and easy to use, but powerful and flexible enough for complex applications.

None of these features is new in themselves. Formal specifications in Z are declarative and structural; and model-checking languages such as SMV are analyzable [1]. What is new is the combination of features, especially of declarative modeling and analyzability.

3.2 Basic Components of Alloy

3.2.1 Atoms

The structures in our models will be built from relations and atoms. An atom is a primitive entity that is indivisible (cannot be broken down into smaller parts), immutable (its

properties don't change over time), and uninterpreted (it doesn't have any built-in properties).

3.2.2 Relations

A relation is a structure that relates atoms [1]. Mathematically, it's a set of tuples, each tuple consisting of a sequence of atoms. In Alloy, every expression denotes a relation. So there won't be any sets of atoms, they'll be represented by unary relations; and there won't be any scalars, they'll be represented by singleton unary relations –that is, relations with one column and one row.

Relations are used to model many different structural features such as containment, labeling, grouping, and linking.

3.2.3 Operators

Expressions in Alloy are constructed by nested applications of operators to variables. All expressions denote relations, so every operator takes one or more relations and yields a relation.

Union, intersection and difference are set operators, they are written in ASCII form: +(union), &(amp;intersection), and -(difference). Each of these operators expects its arguments to have the same type.

The quintessential relational operator is composition or join. The join $p.q$ of relations p and q is the relation you get by taking every combination of a tuple in p and a tuple in q , and including their join, if it exists.

The transpose $\sim r$ of a relation r takes the mirror image of a relation, forming a new relation by reversing the order of each tuple. The transitive closure $\wedge r$ of a binary relation r is the smallest relation that contains r and is transitive. The reflexive transitive closure $*r$ is the

smallest relation that contains r and is both transitive and reflexive, so it's just like \hat{r} but includes additionally a mapping from every atom to itself.

The product $p \rightarrow q$ of two relations p and q is the relation you get by taking every combination of a tuple from p and a tuple from q and concatenating them.

3.2.4 Formulas

Larger formulas are made from smaller formulas by combining them with the standard logical operators, and by quantifying formulas that contain free variables over bindings.

Operators. Alloy uses the standard logical operators, written in programming-language form: '&&' (and), '||' (or), and '!' (not). There are two elementary formulas: $s \text{ in } t$, which says that the expression s denotes a subset of the expression t (or membership when s is a scalar), and $s = t$, which says that the expressions denote the same set. Logical operator \rightarrow denotes implication.

Quantifiers. The existential and universal quantifiers are written *some* and *all*. Less conventionally, '*no* x / F ' and '*sole* x / F ' mean that there is no x and at most one x that satisfies F .

3.3 Structure of Alloy

3.3.1 General Structure

An Alloy model is basically divided into the following parts:

- Signature declaration;
- Invariants: introduced by the keyword *fact*;
- Functions: parameterized formulas, introduced by the keyword *fun*;

- Assertions: specified formulas that are intended to be valid, or putative theorems to be check;
- Commands: ask the Analyzer to do simulation and checking.

More detailed discussion of Alloy structure will be given below, along with analysis of the ABS model.

4. Alloy Model of the ABS

The Alloy model of ABS (Figure 2), based on the description in section 2, focuses on the objects and their interrelations in the system, and the interactions between different kinds of object classes. It also captures the constraints imposed on the whole system.

Figure 2 shows the Alloy model of the ABS. As other Alloy models, the ABS model can basically be divided into four parts: declaration, invariants, functions, and assertions. Following discussion on each part is given in more detail.

- *Declaration.* There are totally 6 signature classes in this model, each of which can be thought of as an object class in JAVA. They are also related to relevant objects in the ABS, ie Badge is related to active badges, Room is related to different areas in a building, Level is associated to the level assigned to each room for access-control, and ABS is related to the whole system itself, etc. What is worth to pay more attention to is that Alloy supports signature extension. This idea is exactly the same as polymorphism in object-oriented programming languages. *sig Entrance extends Room { }* uses signature extension, which means that *Entrance* is a type of *Room* while it can has its own properties that are not common to other types of *Room*.

```

module ActiveBadgeSystem

----- signature declaration -----
sig Level { }

sig Badge {
  accessLevel: set Level,
  signal: option Base,
  -- this implies that signal will be received by no more than one base
  history: set Room
}

sig Base { }

sig Room {
  level: Level,
  bases: set Base
}

sig Entrance extends Room { }
{
  --all Badges have access to the building
  all abs: ABS, b: Badge, e: Entrance | e.Room$level in b.accessLevel
}

sig ABS {
  registered: set Badge,
  rooms: set Room,
  entrance: set Entrance,
  location: registered ->! Room,
  prev : option ABS
}

----- invariants -----

fact nontrivial {
  some Room
  some Entrance
  some Badge
  some ABS
}

fact roomHasBasesAndControlLevel{
  all r: Room | some r.bases && one r.level
}

fact noSharedBase {
  all s: Base | one s.~bases
}

fact signalReceivedByBaseInAccessLevel {
  all abs: ABS, b: Badge, s: Base, r: Room |
    b.signal.~bases.level in b.accessLevel
}

fact locatedByBase {
  all abs: ABS, b: Badge, s: Base, r: Room |
    (b -> r) in abs.location iff s in r.bases && b.signal = s
}

fact registeredBadgesHaveRegistered {
  all abs': ABS, b: Badge | some e: Entrance, some abs1,abs2: ABS |
    b in abs'.registered iff (register(abs1,abs2,b,e) && abs1 in abs'.^prev
    && abs2 in abs'.*prev)
}

```

```

fact stateChangedByMoveRegisterLeave {
    all a1, a2 : ABS |
        a1 in a2.prev iff some b : Badge, r1, r2 : Room, e: Entrance |
            (move(a1, a2, b, r1, r2) || register(a1,a2,b,e) ||
             leave(a1,a2,b,e))
}

fact noSelfPrev {
    all a: ABS | a !in a.prev
}

fact prevNotSymmetric {
    all a1,a2: ABS | a1 in a2.prev && a2 in a1.prev => a1=a2
}

----- functions -----

fun move (abs,abs': ABS, b: Badge, from,to: Room) {
    (b -> from) in abs.location &&
    to.level in b.accessLevel &&
    abs'.location = abs.location ++ (b -> to) &&
    b.history = b.history + to &&
    abs'.prev = abs && abs'.registered=abs.registered &&
    abs'.rooms=abs.rooms && abs'.entrance=abs.entrance
}

fun find (abs: ABS, r: Room, b: Badge) {
    r = b.(abs.location)
}

fun withOthers (abs: ABS, b1: set Badge, b2: Badge) {
    b1 = b2.(abs.location).~(abs.location)
}

fun look (abs: ABS, b: set Badge, r: Room) {
    b = r.~(abs.location)
}

fun register (abs,abs': ABS, b: Badge, e: Entrance) {
    abs'.registered = abs.registered + b &&
    abs'.location = abs.location ++ (b -> e) &&
    b.history = e &&
    abs'.prev = abs && abs'.rooms=abs.rooms && abs'.entrance=abs.entrance
}

fun leave (abs,abs': ABS, b: Badge, e: Entrance) {
    e = b.(abs.location) &&
    abs'.registered = abs.registered - b &&
    abs'.location = abs.location - (b -> b.(abs.location)) &&
    abs'.prev = abs && abs'.rooms=abs.rooms && abs'.entrance=abs.entrance
}

fun show() {
    #ABS.registered=1
    #Room=1
    #Level=1
}

```

```

----- assertions -----
assert BadgeHasAtMostOneLocation {
  all abs: ABS, b: Badge | sole b.(abs.location)
}

assert historyConsistency {
  all abs: ABS, b: Badge, r: Room | b in abs.registered =>
  (b.history).level in b.accessLevel
}

assert moveWorks {
  all abs,abs': ABS, b: Badge, r1,r2: Room |
  move(abs,abs',b,r1,r2) => find(abs',r2,b) && r1+r2 in b.history
}

assert findAndWithWork {
  all abs: ABS, b1,b2: Badge, r: Room, b3: set Badge |
  find (abs,r,b1) && find (abs,r,b2) && withOthers (abs,b3,b2) => b1 in b3
}

assert findAndLookWork {
  all abs: ABS, b1: Badge, b2: set Badge, r :Room |
  find(abs,r,b1) && look(abs,b2,r) => b1 in b2
}

assert lookAndWithWork {
  all abs: ABS, b1: Badge, b2: set Badge, r :Room |
  withOthers(abs,b2,b1) && find(abs,r,b1) => look(abs,b2,r)
}

assert withOthersIsSymmetric {
  all abs: ABS, b1,b2: Badge, bs1,bs2: set Badge |
  withOthers(abs,bs1,b2) && withOthers(abs,bs2,b1) && b1 in bs1 && b1!=b2
  => b2 in bs2
}

assert registerWorks {
  all abs1: ABS, b: Badge, r: Room | some abs2,abs3: ABS, e: Entrance |
  find(abs1,r,b) iff register(abs2,abs3,b,e)
}

assert leaveWorks {
  all abs': ABS, b: Badge | some e: Entrance | some abs: ABS |
  leave(abs,abs',b,e) => no r: Room | r in abs'.rooms &&
  find(abs',r,b)
}

assert registerAndMove {
  all abs': ABS, b: Badge, e: Entrance, r: Room | some abs1,abs2: ABS |
  register(abs1,abs2,b,e) && move(abs2,abs',b,e,r) => e+r in
  b.history && find(abs',r,b)
}

assert sameHistoryImpliesWereTogether {
  all disj b1,b2: Badge, a2: ABS |
  b1+b2 in a2.registered && b1.history=b2.history =>
  some a1: ABS, bs: set Badge |
  a1 in a2.^prev && withOthers(a1,bs,b1) && b2 in bs
}

```

Figure 2. the ABS model in Alloy

Each signature may also include declaration of fields. Fields in signatures describe the interconnections between signatures. In the ABS model, signature *Badge* has three fields: *accessLevel* (the level that it is allowed to access), *signal* (to the base of sensor that receives its signal), and *history* (the history of locations that it has been to). Signature *Room* declares two fields: *level* (the security level this room belong to), and *bases* (the bases it contains). As well, the whole system *ABS* declares five fields: *registered* (all the badges that have registered in this system), *rooms* (all the areas that are controlled by this system), *entrance* (entrances and exits of the building, for simplicity, they were put together here), *location* (the information of each badge's location that the system maintains), and *prev* (the previous state of the system, the keyword *option* was used here is because the initial state of the ABS does not have a previous state).

- *Invariants*. According to the properties of ABS described in section 2, the ABS model is constrained by the following invariants, as introduced by the keyword *fact* in the Alloy model:
 1. *nontrivial*: force the model to be analyzed in a non-trivial way, by defining the fact that there exists some Rooms, Entrances, Badges in the model, and some states of the ABS. This also means that the scopes of these signatures will not be zero.
 2. *roomHasBasesAndControlLevel*: models a constraint that each room has at least one base, and each room is associated with an access control level.
 3. *noSharedbase*: describes the fact that every base belongs to exactly one room.

4. *signalReceivedByBaseInAccessLevel*: models a constraint on the system that the signal transmitted by a Badge will only be received by the base which belongs to a room that is in that Badge's access-control level.
 5. *locatedByBase*: a Badge is located if and only if its signal has been detected.
 6. *registeredBadgesHaveRegistered*: all the registered badges in the system have registered before (the *register* function will be discussed below).
 7. *stateChangedByMoveRegisterLeave*: the state of the ABS can be changed by either the move of a Badge, the registration of a new Badge, or the leaving of an existing Badge. The functions of *move* and *leave* will also be discussed below.
- *Functions*: functions are parameterized formulas, introduced by the keyword *fun*.

Most of the functions used in this model are written in the form such that they can mimic the operations processed by the ABS. There are 7 boolean functions defined in this model, each of which has the following meaning.

1. *move* (*abs,abs'*: ABS, *b*: Badge, *r1,r2*: Room): is true if Badge *b* move from Room *r1* to Room *r2*, and meanwhile the state of ABS changes from *abs* to *abs'*.
2. *find* (*abs*: ABS, *r*: Room, *b*: Badge): is true if in state *abs*, Badge *b* is in room *r*.
3. *withOthers* (*abs*: ABS, *b1*: set Badge, *b2*: Badge): is true if in state *abs*, Badge *b* is currently with the set of Badges *b1* in the same location
4. *look* (*abs*: ABS, *b*: set Badge, *r*: Room): is true if in state *abs*, the set of Badges *b* are currently in room *r*

5. *register* (*abs,abs'*: *ABS*, *b*: *Badge*, *e*: *Entrance*): is true when a Badge *b* enters the building via some entrance *e* and becomes registered, meanwhile the state of the ABS changes from *abs* to *abs'*.
6. *leave* (*abs,abs'*: *ABS*, *b*: *Badge*, *e*: *Entrance*): similar to *register*(), while this happens when a Badge is leaving from the building.
7. *show* (): to run this function can show an instance of the model that satisfies all the constraints. The Constraint Analyzer can generate such a state diagram.

Signature declaration, invariants, and the above-mentioned functions have formally specified an ABS model in Alloy. Based on such formal specifications, the Alloy Constraint Analyzer can be used to check assertions, which are derived properties of the system or putative theorem to be verified.

5. Analyzing the model

The Alloy Constraint Analyzer is a tool for analyzing object models with a variety of uses. At one end, it acts as a support tool for object model diagrams, checking for consistencies of multiplicities and generating sample snapshots. At the other end, it embodies a lightweight formal method in which subtle properties of behavior can be investigated. Its input language Alloy supports a declarative description of state and behavioral properties, by conjoining constraints. An Alloy model can, therefore, be developed incrementally, with the Alloy Constraint Analyzer investigating whatever has been developed so far [2].

Since Alloy is based on predicate logic, it is not a decidable language, so its constraint analyzer cannot provide a sound and complete analysis. Instead, it conducts a

search within a finite *scope* chosen by the user that bounds the number of elements in each primitive type.

The Alloy Constraint Analyzer's output is either an *instance* – a particular state or transition – or a message that no instance was found in the given scope. When checking an assertion, an instance is a counterexample to the theorem. When exercising an invariant or operation, an instance is a demonstration of consistency.

5.1 An Instance Satisfying the Constraints on the Model

By limiting the number of atoms of some signatures to be exactly one in the `show ()` function, the Analyzer can generate a state diagram of the ABS that satisfies the constraints on the model. The reason why I did not use a more general instance here is because when the scope of atoms becomes large (even at the scope of 3), the state diagram will be very hard to read.

5.2 Checking Assertions

When we are going to design a system, we may want to see that some properties of the system hold. To achieve that, we have to define constraints on the system. But how can we know that whether or not those properties hold finally? The Alloy constraint Analyzer contributes a lot in checking such desired properties. Based on the input of an Alloy model and a given assertion to be check, it can automatically investigate all possible states under a given scope and generate a counterexample if that assertion is not valid, without asking the user to input any state of the system.

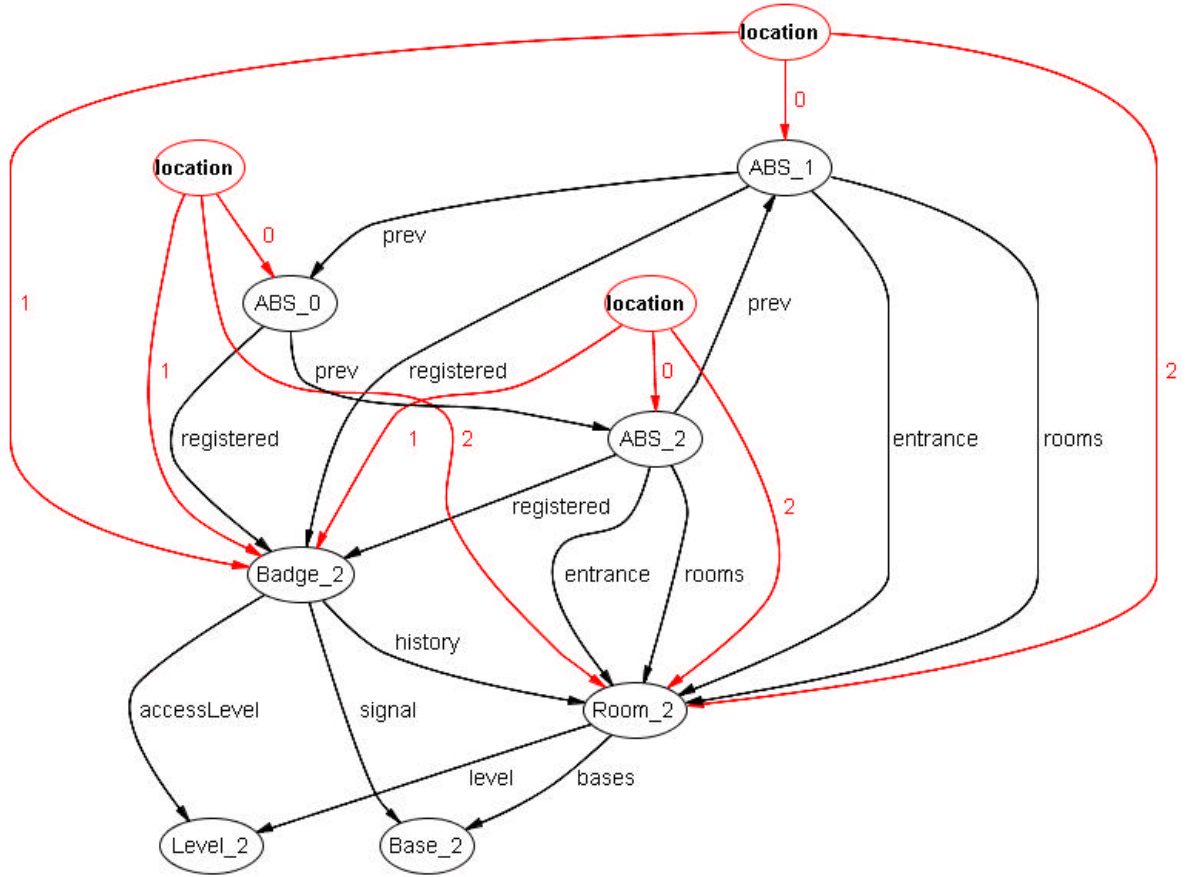


Figure 3. A Trivial Instance that satisfies the constraints

Some properties that should hold in an ABS have been formally written as Alloy assertions. Together with the ABS Alloy model described above, they have been checked by the Constraint Analyzer. Fortunately, these desired properties hold in the scope of 5. In more detail, these assertions are:

1. *BadgeHasAtMostOneLocation*: each Badge has at most one location at any given time, ie. a Badge will not simultaneously be located in two different rooms.
2. *HistoryConsistency*: each Badge's history should not violate the constraint that they should not have been in those areas that they were not allowed to go.

3. *moveWorks*: if a Badge b has moved from Room $r1$ to $r2$, then the ABS should be able to find b at $r2$, and both $r1$ and $r2$ should be in the history list of Badge b .
4. *findAndWithWork*: if the system finds that Badge $b1$ and $b2$ are currently in Room r , and $b2$ is with a set of Badges $b3$ together, then $b1$ should be one of the Badges in $b3$.
5. *findAndLookWork*: if Badge $b1$ is found in Room r , and Room r is found to have a set of Badges $b2$, then $b1$ should be one of the Badges in $b2$.
6. *lookAndWithWork*: if Badge $b1$ is with a set of badges $b2$ and $b1$ is found to be in Room r , then $b2$ should be found in Room r too.
7. *withOthersIsSymmetric*: if $b1$ is with $b2$, then $b2$ is also with $b1$.
8. *registerWorks*: if a Badge b is found in some Room r , then b must have been registered before.
9. *leaveWorks*: if a Badge b has left the building, then b should not be found in the system.
10. *registerAndMove*: if a Badge b registered via some Entrance e , and move to Room r , then b should be found in r and both e and r should be in b 's history.
11. *sameHistoryImpliesWereTogether*: if two Badges $b1$ and $b2$ have the same history in present ABS state, then in some previous state they should have been with each other. This property is really trivial for a real ABS, but it is a good idea to play with.

In all eleven cases, the Alloy Constraint Analyzer completes its search without finding a counterexample. This outcome verifies that such desired properties that the system should hold are consistent with the invariants.

For purpose of demonstration, another trivial assertion that obviously should not hold is also check by the Analyzer. The assertion is given as follow:

```
assert foundAfterMove {  
    all abs,abs' : ABS, b: Badge, r1,r2: Room |  
        move(abs,abs',b,r1,r2) && find(abs',r1,b)  
}
```

It asserts that a Badge b can still be found in Room $r1$ after it has moved from Room $r1$ to $r2$. Such a property is exactly the kind that we do not wish to see in our system. In analyzing the above assertion, the Analyzer gives a counterexample that shows an instance in which the assertion does not hold. In practice, if some undesired properties are unfortunately verified to be held, then one should realize that inconsistencies may exist in model invariants. Review and modification to the model may be required at this point.

After proving that these putative theorems are valid in a reasonable scope, the design of the ABS can then be carried over into implementation, according to constraints imposed on the Alloy model that has been analyzed. This kind of work is beyond the scope of this paper.

6. Conclusions and Summary

Constructing and analyzing a model of the Active Badge System has demonstrated that the design of the ABS based on such specifications is feasible.

One of the most obvious limitations of this Alloy ABS model is that it lacks the capability of representing the sequence of events, ie. it is not able to represent the history of a badge in time order. As mentioned before, software systems have (at least) two kinds of complexity: complexity due to event sequencing, and complexity in the structure of the state

itself. Alloy is designed for structural complexity. If event-sequencing complexity is of great importance to the system, other complementary tools that are good at analyzing sequences of events should be considered.

Alloy is powerful, in a sense that its Constraint Analyzer can investigate all possible state in a given scope to find out an instance that satisfies constraints, or violates an assertion. It has convincing advantages over other object modeling languages on this aspect.

I believe that the use of this kind of lightweight modeling has great benefits, and could result in considerable savings by detecting errors prior to implementation, especially structural flaws that are particularly hard to correct later.

8. Acknowledgement

Dr. Dingel's good advices and kind assistance for my work on this paper are deeply appreciated. I would also like to thank Dr. Dawes' for organizing the course very well.

References

- [1] Daniel Jackson (November, 2001). Micromodels of Software: modeling & Analysis with Alloy. [online]. Available: <http://sdg.lcs.mit.edu/alloy/>. (April 8, 2002).

- [2] Daniel Jackson, Sarfraz Khurshid (n.d). Exploring the Design of an Intentional Naming scheme with an Automatic Constraint Analyzer. [online]. Available: <http://sdg.lcs.mit.edu/~dnj/publications.html> (April 8, 2002).

- [3] Andy Harter, Andy Hopper (November, 1993). A Distributed Location System for the Active Office. [online]. Available: <http://citeseer.nj.nec.com/harter94distributed.html> (April 8, 2002).

- [4] Jon Gibbons, Andy Hopper, Veronica Falcao, Roy Want (n.d). The Active Badge Location System. [online]. Available: <http://citeseer.nj.nec.com/want92active.html> (April 8, 2002).