A survey of Alloy specification language and comparison with an Algebraic specification language

By Chaimanont Thapana

A project paper submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Ogata Kazuhiro

September, 2014

A survey of Alloy specification language and comparison with an Algebraic specification language

By Chaimanont Thapana (1210209)

A project paper submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Ogata Kazuhiro

and approved by
Professor Ogata Kazuhiro
Professor Hiraishi Kunihiko
Associate Professor Aoki Toshiaki

August, 2014 (Submitted)

Abstract

One of the important tasks in the field of software engineering is *software verification*. The tasks of software verification are specifying the system and analyzing whether the system ensures the required properties. To do a software verification, there is an alternative approach that is called *formal specification*, which specifies the system by using a mathematical syntax. At this moment, there are many specification languages that help users to specify system by formal specification, which each one uses different approaches of specification and analysis. One of the interesting specification languages is *Alloy specification language* that uses a new approach for software verification.

This research aims to survey Alloy specification in deeply details about specification and analysis techniques. This report presents the descriptions of relational logic, which is the logic behind specification approach of Alloy, the syntax of Alloy when applying the logic, and the analysis technique, which based on instance finding that relies on SAT solver. So, there are many simple examples in each section of the report, which make the descriptions of Alloy become easier to understand. Moreover, the report also presents an approach of specifying and analyzing the state transition system of concurrent system, which is the case study, in Alloy.

By the way, there are other specification languages that uses the approach of formal specification. One of the other interesting languages is an *algebraic specification language* that is one of the widely used approaches. Exactly, Alloy specification language and an algebraic specification language use the different specification and analysis approaches, which make them have different advantages and disadvantages, and the appropriateness specification in different types of system. So, the comparison of the two different specification languages is the another purpose of this research.

For this research, the algebraic specification language is represented by *Maude specification language*. In the report, there are some descriptions about Maude such as rewriting logic, which is the logic behind specification approach of Maude, and an analysis approach that is a *search* command. In addition, the report also presents about Real-Time Maude that is an extension of Standard Maude and a tool for specifying and analyzing real-time and hybrid systems.

Comparison of Alloy and Maude is experimented through a non-trivial case study. The selected case study of this research is the *hospital problem*, which is one type of the concurrent system that each process uses the shared resources. This report presents the approaches of specifying and solving hospital problem in both Alloy and Maude (including Standard Maude and Real-Time Maude).

From the experimental results that include experiences of using Alloy and Maude, the report presents the description of comparison of Alloy and Maude in many topics, which including advantages and disadvantages of each one. Moreover, there are some rough guidelines for selecting the specification languages between Alloy and Maude that which one is more appropriate to specify each type of system.

Contents

Acknowledgements **1**

1 Introduction **2**

2 Maude Specification Language **5**
2.1 Standard Maude 5
2.2 Real-Time Maude 7

3 Alloy Specification Language **9**
3.1 Relational Logic 9
3.1.1 Atoms and Relations 10
3.1.2 Operators 10
3.1.3 Constraints 13
3.1.4 Declarations and Multiplicity Constraints 14
3.1.5 Integers and Arithmatic 15
3.2 Language 16
3.2.1 Signatures and Fields 16
3.2.2 Facts, Predicates, Functions, and Assertions 17
3.2.3 Commands and Scope 18
3.2.4 Modules 19
3.3 Analysis 20
3.3.1 Instance Finding 20
3.3.2 The Notion of Scope 20
3.3.3 Analysis Constraints and Variables 21
3.3.4 Outputs of Alloy Analyzer 21

4 A Case Study: Hospital Problem **22**

5 Specification and Solving Hospital Problem **25**
5.1 Designing State Transitions of Each Patient 25
5.2 Specification and Solving Problem in Alloy 27
5.2.1 Specification in Alloy 27
5.2.2 Solving Problem in Alloy 35
5.3 Specification and Solving Problem in Maude 37

5.3.1 Specification in Maude 37
5.3.2 Solving Problem in Maude 44
5.3.3 Using Real-Time Maude 47

6 Results and Comparisons **49**
6.1 Results from Alloy 49
6.2 Results from Maude 52
6.3 Comparisons of Alloy and Maude 53

7 Conclusion and Future Works **57**
7.1 Conclusion 57
7.2 Future Works 58
7.2.1 Solving Hospital Problem in automatic approach 58
7.2.2 Comparing Alloy and Maude in other non-trivial case studies 58

Appendix A Specification of Hospital Problem in Alloy **59**

Appendix B Specification of Hospital Problem in Maude **64**

References **75**

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Ogata Kazuhiro for his acute guidance, encouragement, and unlimited support throughout the duration of my master research. Without his support, this research could not have been completed. In addition, I am indebted to him for giving me this invaluable opportunity to study abroad and providing me the financial support.

I would like to thank Mr. Zhang Min for his feedback and wise advices on my research. The quality of this research was significantly improved because of him.

My last acknowledgements go to my family for their support, unconditional love and vital encouragement throughout my study and throughout my life.

Chapter 1

Introduction

Software engineering is the application of engineering to the design, development, and maintenance of software. Moreover, the field of software engineering has produced an enormous body of work known collectively as *software verification* [BBF+10], whose goal is to assure that software fully satisfies all the expected requirements or properties. The process of software verification should be straightforward. First, we use *a specification technique* to specify the underlying ideas and the essence of software or system to create a model. Then from the model, we can use it to verify the properties of the system by using a technique that is called *an analysis technique*. Unlike the testing technique, the space of cases examined in the analysis technique is usually huge and it therefore offers a degree of coverage unattainable in testing. Moreover, the analysis requires no test cases. The user instead provides a property to be checked, which can usually be expressed as succinctly as a single test case.

The model that is created by using the specification technique and is used to analyze properties is called *software abstraction*. The software abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple - a system reduced to its essential form. The good software abstraction should capture their underlying ideas so naturally and convincingly that they seem more like discoveries. To create the good abstractions, there is an alternative approach that is called *formal specification* [Lam00]. It is an approach that specifies the abstractions by using a mathematical syntax to define a notation, which chosen for ease of expression and exploration. There are many specification languages that use the approach of formal specification to specify and analyze systems such as CafeOBJ [DFO03], Promela (the modeling language of the model-checker Spin), Maude, and Alloy. So, each specification language has a different technique to specify systems and analyze properties. Furthermore, they have different advantages and disadvantages. Depending on their techniques and efficiency, each language is appropriate for specifying different types of system. However, a recent survey on the use of formal methods [WLBF09] found that it is nearly impossible for a potential user to decide that which one best matches the system at hand because it depends on the nuances of the system, and prior experience in using these approaches, which can take years to master.

The goal of this research is learning Alloy specification language [Jac12], which uses a new approach for software verification, and is used in many projects such as [GCKP08],

[SyF08], and [LXLZ09]. Alloy is a lightweight formal method [Jac01]. It was inspired by the Z specification language [zsp06] and also strongly influenced by object modeling notations. It takes from formal specification the idea of a precise and expressive notation based on a tiny core of simple and robust concepts. The specification technique of Alloy is based on relational logic, which combines the first-order logic [Smu68] with the relational calculus [Mai83]. For the analysis technique of Alloy, there is the *Alloy Analyzer* that bears little resemblance to model checking, its original inspiration. Instead, it relies on recent advances in SAT (boolean satisfiability) technology [GKSS08]. The Alloy Analyzer translates constraints to be solved from Alloy into boolean constraints, which are fed to an off-the shelf SAT solver. As solvers get faster, so Alloy's analysis gets faster and scales to larger problems. For more descriptions of the techniques of Alloy, they are presented in Chapter 3.

By the way, there are other approaches for software verification. One of the approaches is *an algebraic specification* [EM11]. So, these two different approaches (Alloy specification and algebraic specification) must have their own advantages and appropriate with different types of system. For the another purpose of this research, we decide to compare the two approaches in many topics.

In this research, the algebraic specification language is represented by Maude specification language. Maude [CDE$^+$11] is one of the famous specification languages that is widely used in many projects such as [DMT98], [PPZ01], and [RV07]. It is a language that supports both equational and rewriting logic computation [ND90] for a wide range of applications. Maude has been influenced in important ways by OBJ3 [GKK$^+$88]; in particular, Maude's equational logic sublanguage essentially contains OBJ3 as a sublanguage. The specification technique of Maude is based on rewriting logic that includes as a sub-logic membership equational logic (an extension of order-sorted equational logic). Moreover, Maude is equipped with model checking facilities: the invariants through search and LTL model checker [CDE$^+$07]. By the way, in this research, only the model checking invariants through search is used. For more descriptions of the techniques of Maude, they are presented in Chapter 2.

To compare between Alloy and Maude, we use a non-trivial case study for specifying and analyzing in both languages. The non-trivial case study that we select is *a problem in hospital*. In this problem, there are two groups of patients. In each group, there three patients. Each patient needs to do a list of activities, which each one require different amount of time and number of nurses. A question of this problem is to find the number of nurses that is sufficient for all patients to do all activities in the given limited time. For more descriptions of the case study, we describe in Chapter 4. At this moment, there is no one who specifies this problem by using Maude and Alloy. So, it is worth specifying this problem in Maude and Alloy for more understanding about them. For the approaches of specifying and solving the hospital problem in Alloy and Maude, they are presented in Chapter 5. Furthermore, the results and comparisons between both languages are presented in Chapter 6.

One potential limitation of this research is that the same person applied both specification languages, one after the other. This means that subjective judgments such as

"intuitive", "natural", or "easy to use" would be biased.

In addition to suggestions for improving research methods, Chapter 7 also discuss the conclusion of this research, its adoption, and the future improvement.

Chapter 2

Maude Specification Language

2.1 Standard Maude

Maude is an algebraic specification language that was developed by a team led by Jose Meseguer at the SRI International and University of Illinois at Urbana-Champaign. Maude is a specification and programming language based on rewriting logic that includes as a sub-logic membership equational logic (an extension of order-sorted equational logic). State machines (or state transition systems) are specified in rewriting logic. Data used in state machines are specified in membership equational logic. States of state machines are expressed as data such as tuples and associative-commutative collections (called soups), and state transitions are described in rewrite rules.

The rewrite rule has the form $l : t \rightarrow t'$, where t, t' are terms of the same kind, which may contain variables, and l is the label of the rule. Intuitively, a rule describes a local concurrent transition in a system: anywhere in the distributed state where a substitution instance $\sigma(t)$ of the lefthand side t is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place. And if many instances of the same or of several rules can be matched in different nonoverlapping parts of the distributed state, then all of them can fire concurrently. An unconditional rule is introduced in Maude with the syntax: `rl [<Label>] : <Term-1> => <Term-2>.`

Basic units of Maude specifications are modules. Some built-in modules are provided such as `BOOL` and `NAT` for Boolean values and natural numbers. The Boolean values are denoted as $true$ and $false$, and natural numbers as 0, 1, 2, . . . as usual. The corresponding sorts are `Bool` and `Nat`. Precisely, there are three sorts for natural numbers `Zero`, `NzNat`, and `Nat` that are for zero, non-zero natural numbers, and natural numbers that may be zero or non-zero, respectively. Sort `Nat` is the super-sort of `Zero` and `NzNat`.

Let us consider a simple system as an example. The system is the mutual exclusion protocol. In the system, there are two processes p and q and one lock (which is used to manage the critical section). The type of value of lock is Boolean value. Each process has three sections - remainder section (RS), enter section (ES), and critical section (CS). For any moment, there exists at most one process in the critical section. Initially, each process is in the remainder section and the value of lock is $false$. From the remainder

section, any process can move to the enter section if the value of lock is $false$. Otherwise, it must wait in the remainder section. When any process is in the enter section, it will enter the critical section and the value of lock is changed to be $true$. When any process exits from the critical section, the value of locked is changed to be $false$ and the process must go to the remainder section again. Let us specify this system, precisely a state machine modeling this system, in Maude.

States of current section of each process and the lock are expressed as `pc[i]: l`, and `locked: b`, respectively, where `i` is a process identifier (the corresponding sort is `Pid`), `l` is a label of current section (the corresponding sort is `Label`) and `b` is a Boolean value. `pc[p]: l`, `pc[q]: l`, and `locked: b` are called observable components. A state of the system is expressed as a soup of those observable components, which is expressed as `(pc[p]: l) (pc[q]: l) (locked: b)`. The initial state is expressed as `(pc[p]: RS) (pc[q]: RS) (locked: false)`. Let `init` be the initial state.

Let I be a Maude variable of sort `Pid`, and B be Maude variable of sort `Bool`.

For any process that the current section is remainder section (RS), the action is described in the following rewrite rule:

```
crl[try] : (pc[I]: RS) (locked: B) => (pc[I]: ES) (locked: B) if B == false .
```

where `try` is the label of the rewrite rule, and this rewrite rule is conditional. The condition is `B == false`. If a given term contains an instance of `(pc[I]: RS) (locked: B)` and the condition holds, the instance is replaced with the corresponding instance of `(pc[I]: ES) (locked: B)`.

For any process that the current section is enter section (ES), the action is described in the following rewrite rule:

```
rl[enter] : (pc[I]: ES) (locked: B) => (pc[I]: CS) (locked: true) .
```

This rewrite rule is unconditional.

For any process that the current section is critical section (CS), the action is described in the following rewrite rule:

```
rl[exit] : (pc[I]: CS) (locked: B) => (pc[I]: RS) (locked: false) .
```

The Maude system is equipped with model checking facilities: the *search* command and the LTL model checker. In this research, the search command is used. Given a state s, a state pattern p and an optional condition c, the search command searches the reachable state space from s in a breadth-first manner for all states that match p such that c holds. Such states are called solutions. The syntax is as follows:

```
search in M : s =>* p such that c.
```

where M is a module in which the specification of the state transition system concerned is described or available. A rewrite expression $t => t'$ can be used in the optional condition c. This checks if t' is reachable from t by zero or more rewrite steps with rewrite rules.

The following search finds all states such that they reachable from `init` and there exists two processes are in the critical section at the same time:

```
search in EXPERIMENT : init =>* (pc[I:Pid]: cs) (pc[J:Pid]: cs) CONFIG .
```

where `EXPERIMENT` is the module in which the specification of the system we have been discussing is available. The search finds 2 solutions, namely a counterexample for the

property. It means that this mutual exclusion protocol cannot guarantee the mutual exclusion property.

Note that although the reachable state space from `init` is bounded, the whole state space is unbounded. The search command can be given as options the maximum number of solutions and the maximum depth of search. If the maximum number n of solutions is given, the search terminates when it finds n solutions. Therefore, even if the reachable state from a given state is unbounded, the search command can be used and may terminate. If the maximum depth d of search is given, only the bounded reachable state space from a given state up to depth d is searched. Hence, the search command can be used as a bounded model checker. These options are not used in this research.

2.2 Real-Time Maude

Real-Time Maude [Olv07], that was developed by Peter Csaba Olveczky from Department of Informatics, University of Oslo, extends Maude and Full Maude languages, and is a tool for the high-level formal specification, simulation, and formal analysis of real-time and hybrid systems. Real-Time Maude emphasizes ease and generality of specification, including support for real-time object-based systems that can be distributed, and where the number of objects and messages can change dynamically.

Real-Time Maude specifies the real-time system by using *real-time rewrite theories*. A real-time rewrite theory is a rewrite theory containing:

· A specification of data sort $Time$ specifying the time domain, which may be discrete or dense. The sort $Time$ should satisfy the axioms of the theory *TIME* which defines time abstractly as an ordered commutative monoid ($Time$,0,+,<) with additional operators.

· A designed sort *GlobalSystem* with no subsorts or supersorts, and a free constructor
`{_} : System -> GlobalSystem`
(for *System* the sort of the state of the system) with the intended meaning that $\{t\}$ denotes the whole system in state t. The specification should contain no non-trivial equations involving terms of sort *GlobalSystem*, and the sort *GlobalSystem* should not appear in the arity of any other function symbol in the specification.

· *Instantaneous rewrite rules*, which are ordinary rewrite rules that model instantaneous change and are assumed to take zero time.

· *Tick rules*, that model elapse of time in a system. Tick rules ave the form
$l : t \xrightarrow{\tau_l} t'$ **if** *cond*,
where τ_l is a term (possibly containing variables) of sort $Time$ denoting the duration of the tick rule. The tick rules advance time the system. The global state of the system should always have the form $\{t\}$, in which case the form of the tick rules ensure that time advances uniformly in all parts of the system.

For the tick rule $l : t \xrightarrow{\tau_l} t'$ **if** *cond*, it is written in Real-Time Maude with syntax

```
crl[l] : {t} => {t'} in time τ_l if cond .
```

The syntax for unconditional tick rule is

```
rl[l] : {t} => {t'} in time τ_l .
```

Let us consider a simple system as an example. The system is a system of clock that counts hours. In the system, the clock is a discrete clock where the time always advances by one time unit in each tick step. Moreover, since the clock counts hours, when the clock reaches 24 it should instead show 0. Let us specify this system in Real-Time-Maude.

A state of the system consist of only one observable component, `clock(h)`, where h is the current hour (the corresponding sort is $Time$). Let N be a Real-Time Maude variable of sort $Time$. For increasing the value of hours, the action is described in the following tick rule:

```
rl [tick] : {clock(N)} => {clock((N + 1) rem 24)} in time 1 .
```

where the system advances time by 1 in each step, with the result that the clock value increases by 1, but if the clock reaches 24 it instead shows 0.

Similar as Standard Maude, Real-Time Maude system is equipped with model checking facilities: timed search command and time-bounded LTL model checker. By the way, in this research, the timed search command is used. The timed search command uses Standard Maude's search capabilities and allows the user to search for states that are reachable in a certain time interval from the initial state. The syntax are the following:

```
(search in M: s =>* p such that c with no time limit.)
(search in M: s =>* p such that c in time ∼ t.)
(search in M: s =>* p such that c in time-interval between ∼ t and ∼′ t′.)
```

where $\sim$ and $\sim'$ are either $<, <=, >$, or $>=$, and t and t' are ground terms of sort $Time$.

The following timed search finds all states such that they reachable from state $\{clock(0)\}$, and there exists the counted hours is 24.

```
(tsearch in EXPERIMENT : {clock(0)} =>* {clock(24)} in time < 1000 .)
```

So, the timed search does not find any state that has the value $clock(24)$.

Chapter 3

Alloy Specification Language

Alloy is a specification language that was developed by a team led by Daniel Jackson at the Massachusetts Institute of Technology in the United States in 1997. It was inspired by the Z specification language and also strongly influenced by object modeling notations. The specification technique of Alloy is based on relational logic. Moreover, there is a tool, Alloy Analyzer, that is used to analyze the model of Alloy.

This chapter describes the approach of Alloy with three key elements: a logic, a language, and an analysis:

- **Logic** provides the building blocks of the language. All structures are represented as relations, and structural properties are expressed with a few simple but powerful operators. States and executions are both describing using constraints, allowing an incremental approach in which behavior can be refined by adding new constraints.

- **Language** adds a small amount of syntax to the logic for structuring descriptions. The language helps to build larger models from smaller ones, and to factor out components that can be used more than once.

- **Analysis** is a form of constraint solving. It offers simulation that finds instances of states that satisfy a given property, and checking that has a counterexample that violates a given property. The search for instances is conducted in a space whose dimensions are specified by a *scope*, which assigns a bound to the number of objects of each type.

3.1 Relational Logic

At the core of every specification language is a *logic* that provides the fundamental concepts. It must be small, simple, and expressive. The specification technique of Alloy is based on *relational logic* that combines the quantifiers of first-order logic with the operators of the relational calculus.

3.1.1 Atoms and Relations

By using the relational logic, all structures in the models of Alloy will be built from *atoms* and *relations*, corresponding to the basic entities and the relationships between them.

Atoms

An atom is a primitive entity that is

· *indivisible* : it can't be broken down into smaller parts;

· *immutable* : its properties don't change over time; and

· *uninterpreted* : it doesn't have any built-in properties.

Relations

A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. We can think that a whole model of Alloy is in a relational database system that has only tables, which each entry in tables is an atom. To represent a set of atoms, we use a table with one column. Moreover, we use a table with two or more columns to represent a relation. A relation can have any number of rows, called its *size*. Any size is possible, including zero. The number of column in a relation is called its *arity*, and must be one or more.

Example 3.1.1.1 . *A set of names, a set of addresses, each of size 3, and a binary relation from names to addresses with size 2 are represented as:*

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
address = {(N0,A1), (N1,A2)}
```

3.1.2 Operators

As in a relational database, relational logic uses operators of the relational calculus for querying atoms or relations from the structural tables to create a new temporary table. There are two categories of operators, *set operators* and *relational operators*.

Constants

In relational logic, there are three constants:

· *none* : empty set

· *univ* : universal set

· *iden* : identity

Note that *none* and *univ*, representing the set containing no atom and every atom, respectively, are unary. The identity relation is binary, and contains a tuple relating every atom to itself.

Set Operators

For the set operators, the tuple structure of a relation is irrelevant; the tuples might as well be regarded as atoms. The set operators are:

$\cdot + $ **(union)** : a tuple is in $p + q$ when it is in p or in q (or both);

$\cdot \,\&$ **(intersection)** : a tuple is in $p\&q$ when it is in p and in q;

$\cdot - $ **(difference)** : a tuple is in $p - q$ when it is in p but not in q;

$\cdot$ *in* **(subset)** : p *in* q is true when every tuple of p is also a tuple of q;

$\cdot = $ **(equality)** : $p = q$ is true when p and q have the same tuples.

These operators can be applied to any pair of relations so long as they have the same arity.

Relational Operators

For the relational operators, the tuple structure is essential: these are the operators that make relations powerful. These operators can be applied to any pair of relations whenever they have the different arity. The relational operators are:

$\cdot \to$ **(arrow product)**

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

Example 3.1.2.1 . *Given the following sets of names and addresses*
`Name = {(N0), (N1)}`
`Addr = {(A0), (A1)}`
We have: `Name` $\to$ `Addr = {(N0,A0), (N0,A1), (N1,A0), (N1,A1)}`.

$\cdot$. **(dot join)**

The quintessential relational operator is composition, or join. To join two tuples $s_1 \to ... \to s_m$ and $t_1 \to ... \to t_n$, we first check that the last atom of the first tuple (that is, s_m) matches the first atom of the second tuple (that is, t_n). If they are the same atom, the result of joining is the tuple that starts with the atoms of the first tuple, and finishes with the atoms of the second, omitting just the matching atom. If not, the result is empty.

The dot join $p.q$ of relations p and q is the relation we get by joining every combination of a tuple in p and a tuple in q.

Example 3.1.2.2 . *Given two following relations* `A` *and* `B`
`A = {(A0,N0), (A0,N2), (A1,N1)}`
`B = {(N0,B0), (N0,B1), (N1,B1)}`
We have: `A.B = {(A0,B0), (A0,B1), (A1,B1)}`.

· [ ] **(box join)**

The box operator is semantically identical to dot join, but takes its arguments in a different order, and has different precedence. The expression $e1[e2]$ has the same meaning as $e2.e1$.

· $\sim$ **(transpose)**

The transpose $\sim r$ of a binary relation r takes its mirror image, forming a new relation by reversing the order of atoms in each tuple.

· $^\wedge$ **(transitive closure)**

A binary relation is transitive if, whenever it contains the tuples $a \to b$ and $b \to c$, it also contains $a \to c$, or more succinctly as a relational constraint: $r.r$ *in* r.

The transitive closure $^\wedge r$ of a binary relation r is the smallest relation that contains r and is transitive. We can compute the transitive closure by taking the relation, adding the join of the relation with itself, then adding the join the relation with that, and so on: $^\wedge r = r + r.r + r.r.r + ...$

The reflexive-transitive closure $*r$ is the smallest relation that contains r and is both transitive and reflexive, and is obtained by adding the identity relation to the transitive closure: $*r = {}^\wedge r + iden$.

Example 3.1.2.3 . *Given a following relation* `A`

```
A = {(A0,N0), (A0,N1), (N0,B1), (B1, C2)}
```

We have:

```
^A = {(A0,N0), (A0,N1), (N0,B1), (B1, C2), (A0,B1), (N0,C2), (A0,C2)}.
*A = {(A0,N0), (A0,N1), (N0,B1), (B1, C2), (A0,B1), (N0,C2), (A0,C2),
      (A0,A0), (N0,N0), (N1,N1), (B1,B1), (C2,C2)}.
```

· :> **and** <: **(domain and range restrictions)**

The restriction operators are used to filter relations to a given domain or range. The expression $s <: r$, formed from a set s and a relation r, contains those tuples of r that start with an element in s. Similarly, $r :> s$ contains the tuples of r that end with an element in s.

· ++ **(override)**

The override $p ++ q$ of relation p by relation q is like the union, except that the tuples of q can replace the tuples of p rather than just augmenting them. Any tuple in p that matches a tuple in q by starting with the same element is dropped. The relation p and q can have any matching arity of two or more.

3.1.3 Constraints

We can use the temporary tables, which are queried by using set operators or relational operators, to define constraints of the model of Alloy. To define larger constraints of the model, relational logic uses *logical operators* and *quantifying constraints* of the first-order logic to combine the small constraints.

Logical Operators

The logical operators that are used in relational logic are similar with the operators used in boolean expressions in programming language. The logical operators are:

· *not* **(negation)**

· *and* **(conjunction)**

· *or* **(disjunction)**

· *implies* **(implication)**

· *iff* **(bi-implication)**

Moreover, there is an *else* keyword that can be used with the implication operator;

`C1` *implies* `F1`
else `C2` *implies* `F2`
else `F3`

says that under condition `C1, F1` holds, and if not, then under condition `C2, F2` holds, and if not, `F3` holds.

Quantification

A quantified constraint takes the form
$Q\ x$: $e \mid F$
where F is a constraint that contains the variable x, e is an expression bounding x and Q is a quantifier.

The forms of quantification in Alloy are:

· *all* x: $e \mid F$ (F holds for every x in e);

· *some* x: $e \mid F$ (F holds for some x in e);

· *no* x: $e \mid F$ (F holds for no x in e);

· *lone* x: $e \mid F$ (F holds for at most one x in e);

· *one* x: $e \mid F$ (F holds for exactly one x in e).

Moreover, there is a *disj* keyword that can restrict the binding only to include ones which the bound variables are disjoint from one another by using the keyword *disj* before the declaration. So,

all disj x, y: $e \mid F$

means that F is *true* for any distinct combination of values for x and y.

In addition, quantifiers can be applied to expressions too:

· *some e* (e has some tuples);

· *no e* (e has no tuples);

· *lone e* (e has at most one tuple);

· *one e* (e has exactly one tuple).

Let Expressions

When an expression appears repeatedly, or is a subexpression of a larger, complicated expression, we can factor it out. The form

let $x = e \mid A$

is short for A with each occurrence of the variable x replaced by the expression e.

3.1.4 Declarations and Multiplicity Constraints

Declarations

A declaration introduces a relation name. A constraint of the form

relation−name : *expression*

is called a *declaration*, and says that the relation named on the left has a value that is a subset of the value of the bounding expression on the right. The bounding expression is usually formed with unary relations and the arrow operator, but any expression can be used.

Set Multiplicities

A declaration can include *multiplicity constraints*, which are sometimes implicit. Multiplicities are expressed with the multiplicity keywords:

· *set e* (any number);

· *one e* (exactly one);

· *lone e* (zero or one);

· *some e* (one or more).

From the form x: m e, it constrains the size of x according to m. For a set-valued bounding expression, omitting the keyword is the same as writing *one* keyword.

Relation Multiplicities

When the bounding expression is a relation and is constructed with the arrow operator, multiplicities can be appear inside it. Suppose the declaration looks like this:
r: A $m \to n$ B
where m and n are multiplicity keywords. Then the relation r is constrained to map each member of A to n members of B, and to map m members of A to each member of B.

Example 3.1.4.1 *Given two sets A and B.*

· r: $A \to B$
says that the relation r maps a set A to a set B;

· r: *some* A
says that r is a nonempty subset of A;

· r: A *lone* $\to$ *some* B
says that the relation r maps a set A to a set B such that each member of B belongs to at most one member of A, and each member of A maps to at least one member of B.

3.1.5 Integers and Arithmatic

To create the integer expressions, we need to use the operators of integer. The following operators can be used to combine integers:

· *plus* (addition);

· *minus* (subtraction);

· *mul* (multiplication);

· *div* (division);

· *rem* (remainder).

And the following to compare them:

· = (equals);

· < (less than);

· > (greater than);

· =< (less than or equal to);

· >= (greater than or equal to).

Moreover, there is an operator # that applied to a relation gives the number of tuples it contains. The #e is an integer representing the number of tuples in the relation denoted by e, and that such expressions can be combined with addition and subtraction, and compared.

3.2 Language

A language for describing software abstractions is more than just a logic. To organize a model, language builds larger models from smaller ones, and to factor out components that can be used more than once. There are also small syntactic details that make a language usable in practice. Finally, there's the need to communicate with an analysis tool, by indicating which analyses are to performed.

3.2.1 Signatures and Fields

Signatures

A signature introduces a set of atoms. The declaration

```
sig A { }
```

introduces a set named A. A signature is actually more than just a set, because it can include declarations of relations, and can introduce a new type implicitly.

A set can be introduced as a subset of another set by using the command `extends` or `in`. The forms

```
sig A1 extends A { }, and
sig B1 in B { }
```

introduce sets named $A1$ and $B1$ that are subsets of A and B, respectively. The signatures $A1$ and $B1$ are called *subsignature* of A and B, respectively. Moreover, the signatures A and B are called *top−level signature*. The difference between the commands `extends` and `in` is the mutually disjoint property. By using the command `extends`, the two subsets of the same set are mutually disjoint. However, by using the command `in`, the two subsets are not necessarily mutually disjoint.

Example 3.2.1.1 *Given the following sets*

```
sig A1 extends A { }
sig A2 extends A { }
sig B1 in B { }
sig B2 in B { }
```

The subset $A1$ and $A2$ are disjoint, but the subset $B1$ and $B2$ may intersect.

A multiplicity keyword placed before a signature declaration constrains the number of elements in the signature's set. The form

```
m sig A { }
```

says that set A has m elements.

Furthermore, there is an abstract signature that has no element except those belonging to its extensions. To declare an abstract signature, the form `abstract sig A { }` is used.

Example 3.2.1.2 *Given the following sets*

```
abstract sig T { }
one sig A, B, C extends T { }
```

says that the set T has only threes elements, A, B, and C.

Fields

Relations are declared as fields of signatures. The form

```
sig A {f: e}
```

introduces a relation f whose domain is A, and whose range is given by the expression e, as if a fact included the declaration formula `f in A -> e`.

The expression e can denote a set with a set multiplicity, and can denote a relation (that is, its arity is two or more) with a relation multiplicity.

Example 3.2.1.3 *Given the following sets*

```
sig A {r: B some -> C}
sig B { }
sig C { }
```

says that the relation r is a ternary relation that each of set A has a field of set B associates with set C, which each member of C belongs to at least one member of B.

3.2.2 Facts, Predicates, Functions, and Assertions

The constraints of a model are organized into paragraphs. Assumptions are placed in *fact* paragraphs; implications to be checked are placed in *assertions*; constraints to be used in different contexts are packed as *predicates*; and reusable expressions are packaged as *functions*.

Facts

Constraints that are assumed always to hold are recorded as *facts*. A model can have any number of facts labeled by using the keyword *fact* with the syntax

```
fact "name" { "expressions" }
```

where "*name*" is the name of the fact, and "*expressions*" is the list of expressions that define constraints of a model.

Functions and Predicates

There are constraints that we don't want to record as facts. We might want to analyze the model with a constraint included and excluded; check whether a constraint follows from some other constraints; or declare a constraint so it can be reused in different contexts. Predicates package expressions for such purposes. Functions package expressions for reuse.

A *function* is a named expression, with zero or more declarations for arguments, and an expression bounding for the result. When the function is used, an expression must be provided for each argument; its meaning is just the function's expression, with each argument replaced by its instantiating expression. Functions are defined by using the keyword *fun* with the syntax

```
fun "name" ("parameters"): "returning expression" { "expression" }
```

where "*name*" is the name of the function, "*parameters*" are the list of parameters, "*returning expression*" is the expression that bounds returning value, and "*expression*" is the expression of the function.

A *predicate* is a named constraints, with zero or more declarations for arguments. When the predicate is used, an expression must be provided for each argument; its meaning is just the predicate's constraint with each argument replaced by its instantiating expression. Predicates are always used to define operators of the models. Predicates are defined by using the keyword *pred* with the syntax

```
pred "name" ("parameters") { "expressions" }
```

where "*name*" is the name of the predicate, "*parameters*" are the list of parameters, and "*expressions*" is the list of expressions that define constraints of the predicate.

Assertions

An *assertion* is a constraint that is intended to follow from the facts of the model. The analyzer checks assertions. If an assertion does not follow from the facts, then either a design flaw has been exposed, or a misformulation. Assertions are defined by using the keyword *assert* with the syntax

```
assert "name" { "expressions" }
```

where "*name*" is the name of the assertion, and "*expressions*" is the list of expressions that define constraints of the assertion.

3.2.3 Commands and Scope

To analyze a model, we write a *command* and instruct the tool to execute it. A *run* command tells the tool to search for an instance of a predicate. A *check* command tells it to search for a counterexample of an assertion. The forms

```
run "predicate's name"
check "assertion's name"
```

are used to define the *run* and *check* commands, respectively.

In addition to naming the predicate or assertion, we may also give a *scope* that bounds of the instances or counterexamples that will be considered. If we omit the scope, the tool will use the default scope in which each top-level signature is limited to three elements.

To specify a scope explicitly, we can give a bound for each signature that corresponds to a basic type. We can give bounds on top-level signatures, or on extension signatures, or even on a mixture of the two, so long as whenever a signature has been given a bound, the bounds of its parent and of any other extensions of the same parent can be determined.

Example 3.2.3.1 *Given these declarations of sets*

```
abstract sig B { }
sig B1 extends B { }
sig B2 extends B { }
sig C extends B2 { }
```

and an assertion A, the following commands are well formed:

```
check A for 5 B
check A for 4 B1, 3 B2
```
but this command is ill-formed
```
check A for 3 B1, 3 C
```
because it leaves the bound on B2 unspecified.

Note that a scope declaration only gives an upper bound on the size of each set. If we want to prescribe the exact size, we can use the keyword *exactly*

3.2.4 Modules

Module Declarations and Imports

The first line of every module is an optional *module header* of the syntax
```
module "modulePathName"
```
Every external module that is used must have an explicitly import following the optional header and before any signatures, paragraphs or commands, whose the syntax is
```
open "modulePathName"
```

Parametric Modules

A module can be *parameterized* by one or more signature parameters, given as a list of identifiers in brackets after the module name. Any importing module must then instantiate each parameter with the name of a signature. To import the parametterized modules, the following syntax is used:
```
open "modulePathName" [ "parameters" ]
```
The most common use of parameterized modules is for generic data structures, such as orderings, queues, lists, and trees. The type parameters represent the types of the elements held in the data structure.

Module Ordering

The module *Ordering* is one of the useful modules in Alloy. It is a generic built-in module, which is a parameterized module. It sets the properties between each atom in the set that is a parameter to have an ordering. To use the module *Ordering*, we must import:
```
open util/ordering["Name of Signature"]
```
Moreover, there are useful functions in module *Ordering*. The functions are

· *first* (gives the first element in the order);

· *next* (gives the element following a given element);

· *last* (gives the last element in the order);

· *prevs* (gives the set of all elements that are before a given element).

Example 3.2.4.1 . *Given the signature A that is a parameter of module Ordering*

```
open util/ordering[A]
sig A { }
```

And given the assertion AST that is executed by defining the bound of 3 for signature A

```
assert AST { }
check AST for 3 A
```

In this case, there are 3 atoms in the set A that each one has an order. For example, if the set A has members {$a1$, $a2$, $a3$,}*, one possible order is that* $a1$ *comes before* $a2$ *and* $a2$ *comes before* $a3$*. From this order, the result of the following functions are:*

```
first = {(a1)};
last = {(a4)};
a2.next = {(a3)};
a3.prevs = {(a1), (a2)}.
```

3.3 Analysis

For Alloy specification language, there is a tool for analyzing properties of models of Alloy, that is called *Alloy Analyzer*. The analysis underlying Alloy Analyzer is *instance finding*, which relies on *SAT solver*.

3.3.1 Instance Finding

Checking an assertion and running a predicate reduce to the same analysis problem: finding some assignment of relations to variables that makes a constraint true. So rather than referring to both problems, we'll refer just to the problem of checking assertions.

The *instance finding* is an analysis technique that looks for a refutation, by checking the assertion against a huge set of test cases, each being a possible assignment of relations to variables. If the assertion is found not to hold for a particular case, that case is reported as a *counterexample*. If no counterexample is found, it's still possible that the assertion does not hold, and has a counterexample that is larger than any of the test cases considered.

Moreover, the instance finding is well suited to analyzing invalid assertions because it generates counterexamples, which can usually be easily traced back to the problem in the description, and because invalid assertions tend to be analyzed much more quickly than valid ones (since a valid assertion requires the entire space of possible instances to be covered, whereas, for an invalid assertion, the analysis can stop when the first instance has been found).

3.3.2 The Notion of Scope

To make instance finding feasible, a *scope* is defined that limits the size of instances considered by specifying the number of atoms in each set. The analysis effectively examines every instance within the scope, and an invalid assertion will only not be found if its

smallest counterexample is outside the scope. A richer notion of scope turns out to be more useful , in which each signature is bounded separately, under the user's control.

The scope thus defines a multidimensional space of test cases, each dimension corresponding to the bound on a particular signature. Even a small scope usually defines a huge space. In the default scope of 3, for example, which assign a bound of three to each signature, each binary relation contributes 9 bits to the state (since each three elements of the domain may or may not be associated with each three elements of the range) - that is, a factor of 512. So a tiny model with only four relations has a space of over a billion cases.

Instance finding has far more extensive coverage than traditional testing, so it tends to be much more effective at finding bugs. In short: *Most bugs have small counterexamples.* That is, if an assertion is invalid, it probably has a small counterexample. It is called the *"small scope hypothesis"* by Daniel Jackson [JV00], and it has an encouraging implication: if we examine all small cases, we are likely to find a counterexample.

3.3.3 Analysis Constraints and Variables

When we run a predicate or check an assertion, the analyzer searches for an instance of an analysis constraint: an assignment of values to the variables of the constraint for which the constraint evaluates to true.

In the case of a predicate, the analysis constraint is the predicate's constraint conjoined with the facts of the model. An instance is an example: a scenario in which both the facts and the predicate hold.

In the case of an assertion, the analysis constraint is the negation of the assertion's constraint conjoined with the facts of the model. An instance is a counterexample: a scenario in which the facts hold but the assertion does not (or, equivalently, a scenario in which the assertion fails to follow from the facts).

The variables that are assigned in an instance comprise:

· the sets associated with the signatures;

· the relations associated with the fields;

· for a predicate, its arguments.

3.3.4 Outputs of Alloy Analyzer

When executing the *run* command or *check* command, if Alloy Analyzer can find an instance or a counterexample, it will show only one possible instance. The tool's selection of instances is arbitrary, and depending on the preferences we have set, may even change from run to run. In practice, though, the first instance generated does intend to be a small one. This is useful, because the small instances are often pathological, and thus more likely to expose subtle problems.

Moreover, outputs from Alloy Analyzer can be shown in a variety of forms, textual and graphical, that makes it easy to understand the results.

Chapter 4

A Case Study: Hospital Problem

The hospital problem is one of the non-trivial case studies that were not specified and solved by using Alloy and Maude before. So, we decide to use the hospital problem as a case study to compare between Alloy and Maude. Let us describe the case study in more details.

In the hospital problem, there are two groups of patients, $G1$ and $G2$, and each group consists of three patients. Moreover, in the hospital, there are six rooms as follows:

· **Room 1** ($R1$) : the room for patients in the group $G1$;

· **Room 2** ($R2$) : the room for patients in the group $G2$;

· **Rehabilitation Room** (RR) : the room for patients to do a rehabilitation;

· **Bath Room** (BR) : the room for patients to take a bath.

For each patient, he needs to do activities that each one requires the different number of nurses to help, and limitation of time to do. The patients from the same group need to do the same activities with the same requirements. However, for the patients from the different groups, they need to do the different activities and different requirements to do each activity.

For each patient in the group $G1$, he starts with having a meal in the room $R1$, moving to the rehabilitation room RR, doing a rehabilitation, and finally, coming back to the room $R1$. So, there are the constraints for doing activities as follows:

· For each patient to have a meal in room $R1$, it takes time 30 to 60 minutes and requires 1 nurse supporting his meal.

· For each patient to move from room $R1$ to the rehabilitation room RR, it takes time 3 to 5 minutes and requires 1 nurse helping to move.

· For each patient to do a rehabilitation in room RR, it takes time 30 to 45 minutes and requires no nurse.

· For each patient to come back from the rehabilitation room RR to the room $R1$, it takes time 3 to 5 minutes and requires 1 nurse helping to move back.

· For the rehabilitation room RR, at most 2 patients can do a rehabilitation at the same time.

For each patient in the group $G2$, he starts with having a meal in the room $R2$, moving to the bath room BR, taking a bath, and finally, coming back to the room $R2$. So, there are the constraints for doing activities as follows:

· For each patient to have a meal in room $R2$, it takes time 30 to 60 minutes and requires no nurse.

· For each patient to move from room $R2$ to the bath room BR, it takes time 1 to 5 minutes and requires 2 nurses helping to move.

· For each patient to take a bath in room BR, it takes time 15 to 30 minutes and requires 1 nurse helping to take a bath.

· For each patient to come back from the bath room BR to the room $R2$, it takes time 1 to 5 minutes and requires 2 nurses helping to move back.

· For the bath room BR, at most 2 patients can take a bath at the same time.

Figure 4.1 and Figure 4.2 represent the sequence of activities, and the requirements of nurses and time of each activity for the patients from groups $G1$ and $G2$, respectively.

The question of the hospital problem is that when the six patients can start having meals at a same time, how many nurses that are sufficient for finishing the aforementioned activities, which satisfy the following requirements:

· Those activities should be done in 180 minutes;

· Each patient from group $G2$ should start moving back to room $R2$ from the bath room BR in 5 minutes (inclusive) after finish taking a bath (says that we should not keep them waiting in bath room BR more than 5 minutes after finish taking a bath).

Note that there are two assumptions for the hospital problem. The assumptions are

· It does not take any time for a nurse alone to move from one place to another;

· The times are discrete, which can be expressed as integers or natural numbers.

By the way, we can think that the whole system of hospital problem is the concurrent system that has shared resources [Bac93]. We can see the six patients as six processes that concurrently execute in the system. For nurses, spaces in rehabilitation room RR, and spaces in bath room BR, we can see them as the shared resources that all processes use.

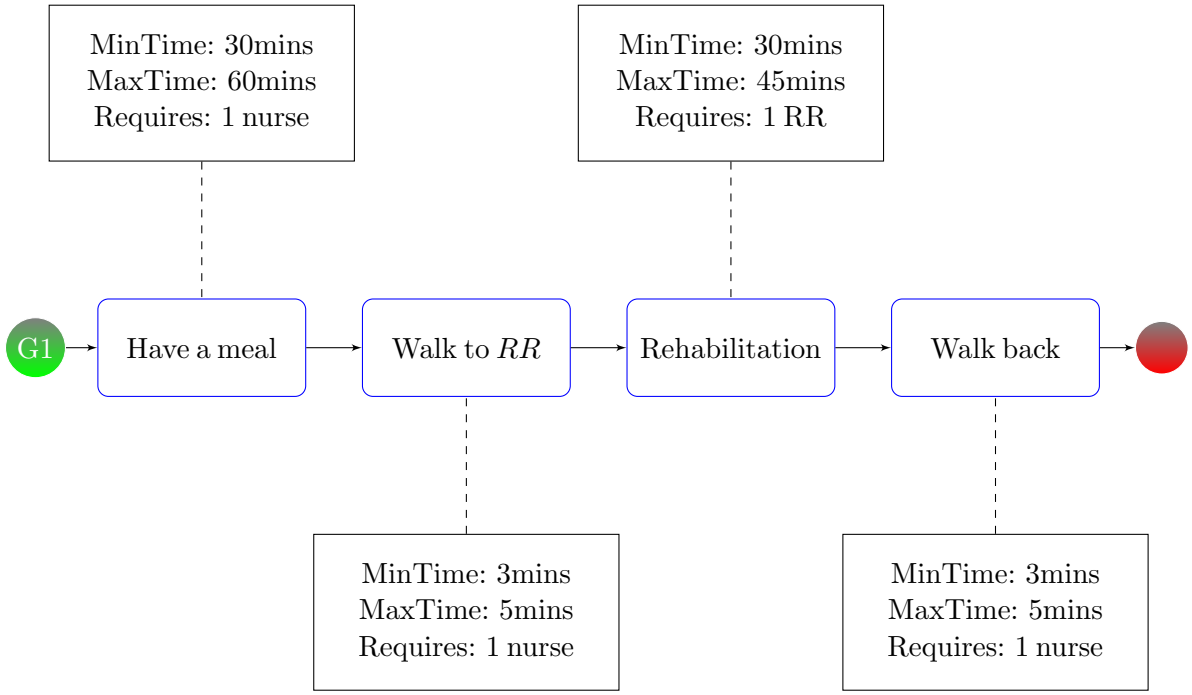

Figure 4.1: Diagram represents activities that each patient in group $G1$ needs to do.

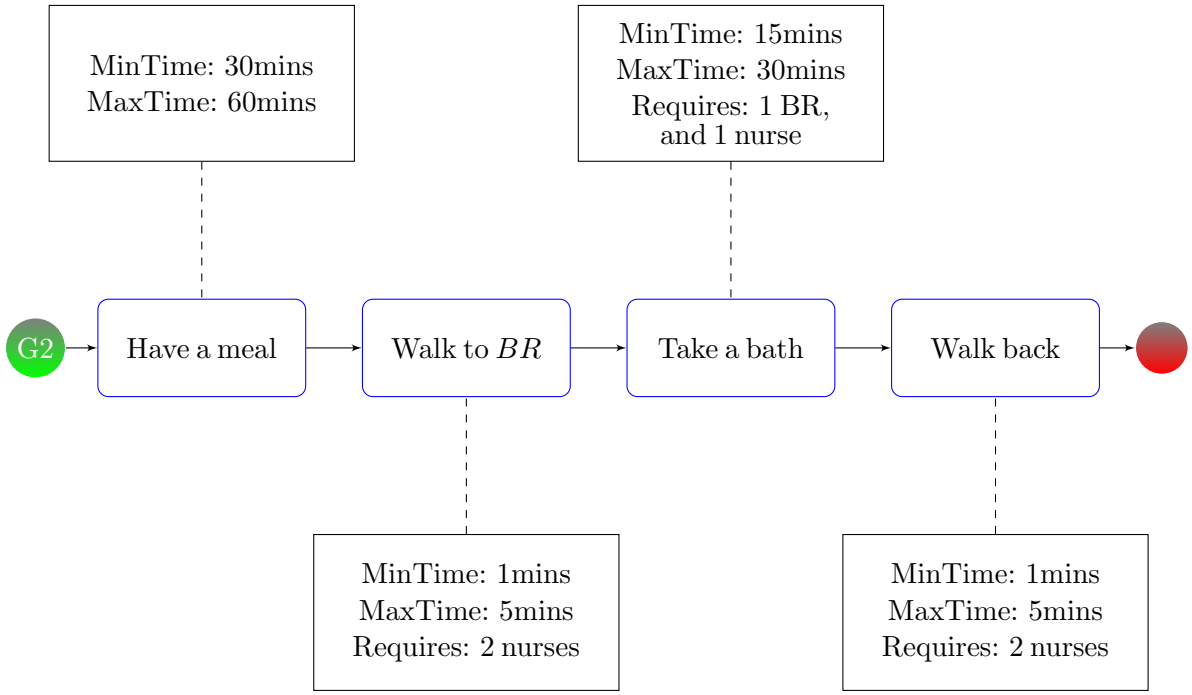

Figure 4.2: Diagram represents activities that each patient in group $G2$ needs to do.

Chapter 5

Specification and Solving Hospital Problem

In this research, we specify the hospital problem in both specification languages, Alloy and Maude.

5.1 Designing State Transitions of Each Patient

To specify the hospital problem, first, we need to design the state transitions that represent the steps of doing activities for each patient in two groups. The steps of doing activities for patients in groups $G1$ and $G2$ are represented with the same state transitions (just some parts are difference). So rather than referring to both groups of patients, we'll refer just to each patient in group $G1$.

Figure 5.1 shows the state transition that represents steps of doing activities for each patient in group $G1$. There are 9 states, which can be separated into two groups. The first group, we call group of *Doing* states, consists of $Meal$ state, $WalkToRR$ state, $InRR$ state and $WalkBackR1$ state. The group of *Doing* states represents the states that refers to the main activities, that each patient needs to do. For the second group, we call group of $Waiting$ states, consists of *Remain* state, $WaitForWalkToRR$ state, $WaitForInRR$ state, $WaitForWalkBackR1$ state and Fin state. The group of $Waiting$ states represents the states that each patient are waiting for doing the main activities when the resources are not sufficient to do. Each patient may or may not be in the states in the group of $Waiting$ states, but they must be in all states of the group of *Doing* state.

In the initial step, each patient is in the *Remain* state and has a state transition as follows:

· **In the Remain state** : patient must change state to $Meal$ state for having a meal. However, before changing state, it needs to check that can the patient has a meal? The condition to check is that the number of remaining nurses must suffice to help. If so, the patient must move to $Meal$ state. Otherwise, he must wait in *Remain* state until the condition holds.

· **In the Meal state** : patient must take time to have a meal at least 30 minutes, but at most 60 minutes before changing state to $WalkToRR$ state. However, before changing state, it needs to check that can the patient walks? The condition to check is that the number of remaining nurses must suffice. If so, the patient must change to $WalkToRR$ state. Otherwise, he must wait in $WaitForWalkToRR$ state until the number of nurses suffices.

· **In the WaitForWalkToRR state** : patient must always try to change state to $WalkToRR$ state. If the condition to change the state holds, the patient must change the state immediately.

· **In the WalkToRR state** : patient must take time to walk at least 3 minutes, but at most 5 minutes before changing state to $InRR$ state. However, before changing state, it needs to check that can the patient do a rehabilitation? The condition to check is that there is a free space in the rehabilitation room. If so, the patient must change to $InRR$ state. Otherwise, he must wait in $WaitRR$ state until the condition holds.

· **In the WaitForInRR state** : patient must always try to change state to $InRR$ state. If the condition to change the state holds, the patient must change the state immediately.

· **In the InRR state** : patient must take time to do a rehabilitation at least 30 minutes, but at most 45 minutes before changing state to $WalkBackR1$ state. However, before changing state, it needs to check that can the patient walks back? The condition to check is that the number of remaining nurses must suffice. If so, the patient must change to $WalkBackR1$ state. Otherwise, he must wait in $WaitForWalkBackR1$ state until the number of nurses suffices.

· **In the WaitForWalkBackR1 state** : patient must always try to change state to $WalkBackR1$ state. If the condition to change the state holds, the patient must change the state immediately.

· **In the WalkBackR1 state** : patient must take time to walk back at least 3 minutes, but at most 5 minutes before changing state to Fin state.

· **In the Fin state** : if the patient is in the Fin state, it means that the patient has already done all activities, and he must not change state anymore.

Note that when each patient changes the state, he must release all acquired resources before trying to change the state. For example, if the patient is in the $Meal$ state and wants to change the state to $WalkToRR$ state, he must release the resource of one nurse to the system before changing.

For the patients in group $G2$, we use the same state transition to represent their steps of doing activities. However, there are some different points between the state transitions of patients in group $G1$ and $G2$, which are:

· the names of states, which represent the different activities, in state transitions;

· the conditions, which are used to check that can patients change states?;

· the limitation of time that patients must take to do each activity; and

· the resource of spaces in the bath room BR that is used instead of spaces in the rehabilitation room RR.

5.2 Specification and Solving Problem in Alloy

5.2.1 Specification in Alloy

To specify the hospital problem in Alloy, we specify the whole system with atoms and relations that are represented by signatures and fields. Moreover, we describe the constraints and state transitions with facts.

Specification of Atoms and Relations of Hospital Problem

For the hospital problem, there are 3 abstract signatures, 23 signatures, and 3 fields.
The signatures are:

· **Patient** : we specify the signatures of patients as in Program 5.1, from line 4 to line 11. The signature *Patient* is an abstract signature, which consists of three subsignatures. The three subsignatures are:

  · **Patient1** : represents a set of atoms of patients in group $G1$,

  · **Patient2** : represents a set of atoms of patients in group $G2$,

  · **NoPatient** : represents an atom of a dummy patient that does not refer to any patient in the system. This atom is used to associate with nurses that are not helping any patient in each time.

· **Nurse** : we specify the signature of nurses as in Program 5.1, from line 13 to line 15. The signature $Nurse$ represents a set of atoms of nurses.

· **Activity1** : we specify the signature of activity1 as in Program 5.1, from line 17 to line 26. The signature $Activity1$ represents the set of all states that refers to steps of doing activities for each patient in group $G1$. It is an abstract signature that consists of 9 subsignatures. Each subsignature represents each state of activity, and each one consists of only one atom.

· **Activity2** : we specify the signature of activity2 as in Program 5.1, from line 28 to line 37. The signature $Activity2$ represents the set of all states that refers to steps of doing activities for each patient in group $G2$. Same as $Activity1$, it is an abstract signature that consists of 9 subsignatures.

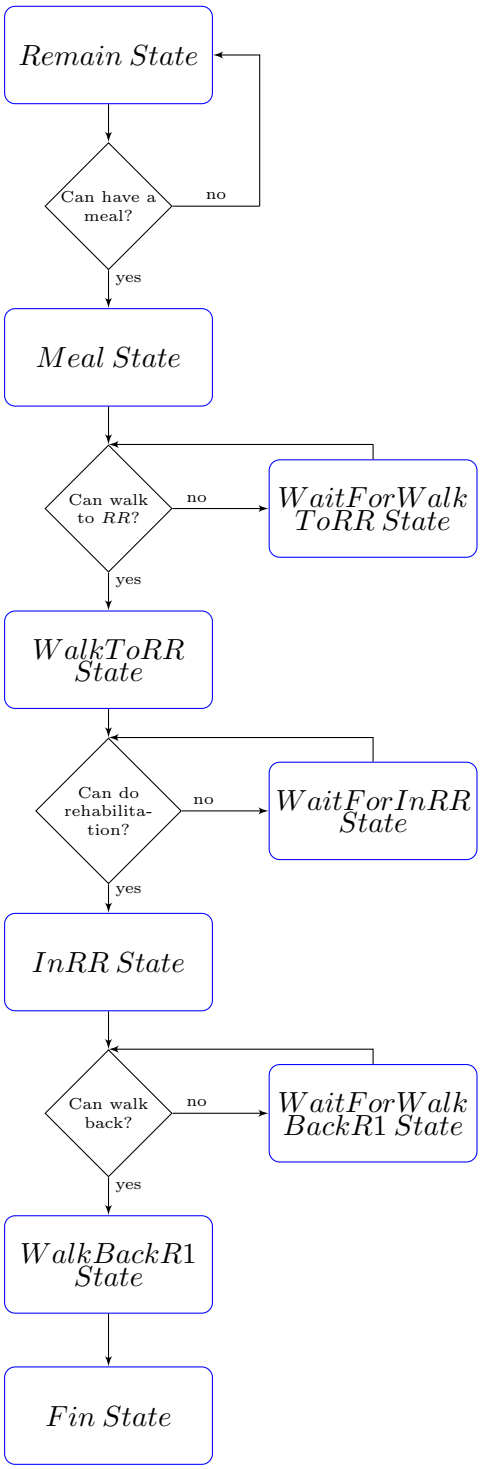

Figure 5.1: Diagram represents state transition of each patient in group $G1$.

· **Time** : we specify the signature of time as in Program 5.1, from line 1 to line 2. Since the signature $Time$ is defined as a parameter of module $Ordering$, the atoms in its set have an order. Each atom in set $Time$ is used to represent each minute of the system.

The fields are:

· **doActivity1** : we specify the field $doActivity1$ in the signature $Patient1$. The field $doActivity1$ is a ternary relation that associating each atom of $Patient1$ with relation from $Activity1$ to $Time$, which each atom of $Time$ belongs to exactly one atom of $Activity1$. This relation represents the activity that each patient in group $G1$ does in each minute. Moreover, for each minute, each patient in group $G1$ must do exactly one activity.

· **doActivity2** : we specify the field $doActivity2$ in the signature $Patient2$. The field $doActivity2$ is a ternary relation that associating each atom of $Patient2$ with relation from $Activity2$ to $Time$, which each atom of $Time$ belongs to exactly one atom of $Activity2$. This relation represents the activity that each patient in group $G2$ does in each minute. Moreover, for each minute, each patient in group $G2$ must do exactly one activity.

· **help** : we specify the field $help$ in the signature $Nurse$. The field $help$ is a ternary relation that associating each atom of $Nurse$ with relation from $Patient$ to $Time$, which each atom of $Time$ belongs to exactly one atom of $Patient$. This relation represents the patient that each nurse helps in each minute. Moreover, for each minute, each nurse must help exactly one patient. However, in the case of nurse that does not help any patient, the nurse must associate with the dummy patient,$NoPatient$, in that minute.

Note that the relations of the system are used to represent the states of system, which record that in each minute, each patient does which activity and each nurse helps which patient?

Example 5.2.1.1 . *Given following sets of* $Patient1$*,* $Nurse$*, and* $Time$*:*

```
Patient1 = {(P0), (P1), (P2)}
Nurse = {(N0), (N1)}
Time = {(T0), (T1), (T2)}
```

If in the relation $doAcitivity1$ *consists of a tuple* $(P0, Meal, T2)$*, it means that at the minute* $T2$*, patient* $P0$ *has a meal. Moreover, if in the relation help consists of a tuple* $(N0, P0, T2)$*, it means that at the minute* $T2$*, Nurse* $N0$ *helps patient* $P0$*.*

Program 5.1: Specification of signatures and fields of Hospital Problem

```
1   open util/ordering[Time]
2   sig Time {}
```

```
3
4   abstract sig Patient {}
5   sig Patient1 extends Patient {
6        doActivity1: Activity1 one -> Time
7   }
8   sig Patient2 extends Patient {
9        doActivity2: Activity2 one -> Time
10  }
11  one sig NoPatient extends Patient {}
12
13  sig Nurse {
14      help: Patient one -> Time
15  }
16
17  abstract sig Activity1 {}
18  one sig Remain extends Activity1 {}
19  one sig Meal extends Activity1 {}
20  one sig WaitForWalkToRR extends Activity1 {}
21  one sig WalkToRR extends Activity1 {}
22  one sig WaitForInRR extends Activity1 {}
23  one sig InRR extends Activity1 {}
24  one sig WaitForWalkBackR1 extends Activity1 {}
25  one sig WalkBackR1 extends Activity1 {}
26  one sig Fin extends Activity1 {}
27
28  abstract sig Activity2 {}
29  one sig Remain2 extends Activity2 {}
30  one sig Meal2 extends Activity2 {}
31  one sig WaitForWalkToBR extends Activity2 {}
32  one sig WalkToBR extends Activity2 {}
33  one sig WaitForInBR extends Activity2 {}
34  one sig InBR extends Activity2 {}
35  one sig WaitForWalkBackR2 extends Activity2 {}
36  one sig WalkBackR2 extends Activity2 {}
37  one sig Fin2 extends Activity2 {}
```

Specification of State Transitions

The state of Alloy is recorded as a tuple in the relations. We use *fact* to define state transitions as the constraints that are assumed always to hold. Moreover, we use *predicate* to define operators that are used to assign the values of variables in the state when changing minutes of time. As we described in the section 5.1, the state transitions that represent step of doing activities of patients in group $G1$ and $G2$ are the same (just the

names and some conditions are difference). So, we'll refer just to the specification of state transitions of each patient in group $G1$.

Program 5.2 represents the specification of state transitions of hospital problem in Alloy. We use *predicate* as operators to assign the values of variables in the states (values of atoms in the tuples in each relation). In this system, there are two roles of *predicate*.

The roles of *predicate* are:

· **Assigning the initial state to all patients** : we specify the predicate that is used to assign the values of initial state as in Program 5.2, from line 1 to line 4.

  The predicate is named *init*, and it has one parameter. The parameter is a given minute of time t, that is set to be an initial minute of the system. In the predicate, it assigns the tuples, which associate each patient in group $G1$ with a relation from atom *Remain* to a given minute t (and each patient in group $G2$ with a relation from atom *Remain*2 to a given minute t), to the relation *doActivity*1 (and *doActivity*2). It represents that in the initial state of the system, all patients must be in the *Remain* state.

· **Assigning the state to a given patient and minute** : in the system, there are 9 predicate paragraphs of this type of predicate for each group of patients (18 predicate paragraphs for two groups of patients). Each one is used to assign each state of doing activity to a given patient at a given minute. Since all 18 predicates are the same (just changes the names of predicates and names of states), we'll show only one predicate paragraph that is specified as in Program 5.2, from line 6 to line 8.

  The predicate is named $haveMeal$, and has two parameters. The parameters are a given minute of time t', and a given patient p in group $G1$. In the predicate, it assigns the tuple, which associate the patient p with a relation from atom $Meal$ to a given time t', to the relation *doActivity*1. It represents that at the given time t', the given patient p must be in the $Meal$ state.

Moreover, we use $fact$, is named $Traces$, to define state transitions as in Program 5.2, from line 10 to line 31.

At first, in line 11, we call the predicate *init* to assign the initial state by parsing the return value of operator $first$ as a parameter. The operator $first$ returns the first atom of order in the set $Time$. And then, we define the constraint for state transition, as from line 12 to 30, that for each minute t, each patient p in group $G1$ must be assigned the next state of activity in the next minute t', which depends on the current state (where t represent a current minute and t' represents a next minute that follows t from the order in set $Time$). To check the current states of activities of each patient in group $G1$, we check the tuples in the relation *doActivity*1 by using *dot join* operator. Moreover, to assign the next states of activities of each patient in group $G1$, we use the predicate that are described above, by parsing the next time t' and each patient p as parameters.

We assign the next states of activities for each patient in state transition depends on the current states. There are 9 cases of the current states as following:

· **Remain state (p.doActivity1.t = Remain)** : the next state can be $Meal$ state, or the same state ($Remain$ state).

· **Meal state (p.doActivity1.t = Meal)** : the next state can be $WaitForWalkToRR$ state, $WalkToRR$ state, or the same state ($Meal$ state).

· **WaitForWalkToRR state (p.doActivity1.t = WaitForWalkToRR)** : the next state can be $WalkToRR$ state, or the same state ($WaitForWalkToRR$ state).

· **WalkToRR state (p.doActivity1.t = WalkToRR)** : the next state can be $InRR$ state, $WaitForInRR$ state, or the same state ($WalkToRR$ state).

· **WaitForInRR state (p.doActivity1.t = WaitForInRR)** : the next state can be $InRR$ state, or the same state ($WaitForInRR$ state).

· **InRR state (p.doActivity1.t = InRR)** : the next state can be $WaitForWalkBackR1$ state, $WalkBackR1$ state, or the same state ($InRR$ state).

· **WaitForWalkBackR1 state (p.doActivity1.t = WaitForWalkBackR1)** : the next state can be $WalkBackR1$ state, or the same state ($WaitForWalkBackR1$ state).

· **WalkBackR1 state (p.doActivity1.t = WalkBackR1)** : the next state can be Fin state, or the same state ($WalkBackR1$ state).

· **Fin state (p.doActivity1.t = Fin)** : the next state can be only Fin state (it will not change the state anymore).

Program 5.2: Some Parts of Specification of State Transitions in Alloy

```
1   pred init (t: Time) {
2    all p: Patient1 | p.doActivity1.t = Remain
3    all q: Patient2 | q.doActivity2.t = Remain2
4   }
5
6   pred haveMeal (t': Time, p: Patient1) {
7    p.doActivity1.t' = Meal
8   }
9
10  fact Traces {
11   first.init
12   all t: Time - last | let t' = t.next |
13    all p: Patient1 |
14     p.doActivity1.t = Remain implies
15      (haveMeal[t',p] or notDo[t',p])
16     else p.doActivity1.t = Meal implies
17      (walkRR[t',p] or waitForWalkRR[t',p] or haveMeal[t',p])
18     else p.doActivity1.t = WaitForWalkToRR implies
```

```
19          (walkRR[t',p] or waitForWalkRR[t',p])
20        else p.doActivity1.t = WalkToRR implies
21          (RR[t',p] or waitForRR[t',p] or walkRR[t',p])
22        else p.doActivity1.t = WaitForInRR implies
23          (RR[t',p] or waitForRR[t',p])
24        else p.doActivity1.t = InRR implies
25          (walkBack[t',p] or waitForWalkBack[t',p] or RR[t',p])
26        else p.doActivity1.t = WaitForWalkBackR1 implies
27          (walkBack[t',p] or waitForWalkBack[t',p])
28        else p.doActivity1.t = WalkBackR1 implies
29          (done[t',p] or walkBack[t',p])
30        else done[t',p]
31    }
```

By using only *fact Traces* to define state transitions, each patient can do any activity without any constraint of time and any requirement. So, we need to specify more constraints of the system that are described in the next section.

Specification of Constraints

We specify the constraints of hospital problem by using *fact*, names *SetConstraints*, that is showed some parts in Program 5.3. For the constraints of hospital problem, there are four types of constraints. The types of constraints are:

· **Constraints of limit of time for doing activity** : this type of constraint is used to specify the minimum and the maximum minutes that each patient must take to do each activity.

  To specify the constraint, we set the minimum and the maximum numbers of tuples in the relations *doAcitivity*1 and *doActivity*2 by using the operator # to count the number of tuples of each state of activities. Since the approach that is used to specify for all states of activities are the same (just change the minimum and the maximum minutes, and names of states of activities), we'll show just to the constraint of state *InRR* of patients in group $G1$, which is specified as in Program 5.3, from line 2 to line 3. The constraint says that for any patient p in group $G1$, the number of tuples, that consist of the atom *InRR*, in the relation *doActivity*1 must be at least 30 tuples and at most 45 tuples.

  Note that the states of activities in the group of *Waiting* states do not need to specify this type of constraint, because there is no limit of time for waiting.

· **Constraints of requiring nurses for doing activity** : this type of constraint is used to specify the number of nurses that is required for helping each patient to do each activity.

  To specify the constraint, we set the number of tuples in the relation *help* by using the operator #. Since the approach that is used to specify for all states of activities

are the same (just change the number of nurses, and names of states of activities), we'll show just to the constraint of state $Meal$ of patients in group $G1$, which is specified as in Program 5.3, from line 5 to line 6. The constraint says that for any patient p, if he is in the $Meal$ state at any minute t, then at the same minute t, the number of nurses that helps the patient p must be 1.

Note that for the state in the group of $Waiting$ states, the number of required nurse is 0.

· **Constraints of the number of patients that can enter the rooms** : In the hospital problem, there are two rooms, a rehabilitation room (RR) and a bath room (BR). For each room, at most two patients can enter in the same minute.

To specify the constraint, we set the maximum numbers of tuples, that consists of the $InRR$ (or $InBR$), in the relation $doActivity1$ (or $doActivity2$) at any time by using the operator #. The constraints are described as in Program 5.3, from line 8 to line 9. It says that for any minute t, the number of tuples, that consists of atom $InRR$ (and $InBR$), in the relation $doActivity1$ (and $doActivity2$) must be less than 3 tuples.

· **Constraints of the states in the group of Waiting state** : Because there is no constraint of limit of time of the states in the group of $Waiting$ states, each patient can stay in the $Waiting$ states all the time, and it enables the flaw of the specification. So, we need to specify this type of constraint. This type of constraint is used to specify that each patient can be in the states of the group of $Waiting$ state only when he cannot do the main activities.

Since the approach that is used to specify for all $Waiting$ states are the same (just change conditions, and names of states of activities), we'll show just to the constraint of state $WaitForWalkToRR$ and $WaitForInRR$, which are specified as in Program 5.3, from line 11 to line 13, and from line 14 to line 16, respectively.

The constraint of $WaitForWalkToRR$ state says that for any patient p in group $G1$, if he is in the $WaitForWalkToRR$ state at any minute t, then at the same minute t the number of available nurses must be less than 1 nurse. Because to walk to the rehabilitation room RR, the patient p requires 1 nurse to help.

The constraint of $WaitForInRR$ state says that for any patient p in group $G1$, if he is in the $WaitForInRR$ state at any minute t, then at the same minute t the number of free spaces in a rehabilitation room RR must be more than or equal 2. Because at most 2 patients can enter the rehabilitation room RR at the same time.

Note that the Fin state and $Fin2$ state do not need to specify this type of constraint, because each patient in group $G1$ and group $G2$ must stay in the Fin state and $Fin2$ state, respectively, after finishing all main activities.

Program 5.3: Some Parts of Specification of Constraints of Hospital Problem in Alloy

```
1 fact SetConstraints {
```

```
2   all p: Patient1 | #(InRR.(p.doActivity1)) > 29 and
3                     #(InRR.(p.doActivity1)) < 46
4
5   all p: Patient1 | all t: Time | p.doActivity1.t = Meal
6                                    implies #((help.t).p) = 1
7
8   all t: Time | #((doActivity1.t).InRR) < 3
9   all t: Time | #((doActivity2.t).InBR) < 3
10
11  all p: Patient1 | all t: Time - first |
12            (p.doActivity1.t = WaitForWalkToRR) implies
13                                 (#((help.t).NoPatient) < 1)
14  all p: Patient1 | all t: Time - first |
15            (p.doActivity1.t = WaitForInRR) implies
16                                 (#((doActivity1.t).InRR) >= 2)
17                                 .
18                                 .
19                                 .
20  }
```

5.2.2 Solving Problem in Alloy

To solve the hospital problem in Alloy, we specify the requirements with *assertion*, and execute it with the *check* command by using Alloy Analyzer.

Specifying the Requirements with Assertion

The requirements of patients in group $G1$ are that each patient must be done all activities in a given time. From our specification of state transitions, any patient p in group $G1$ has already done all activities, if the patient p is in the Fin state. We specify the requirement of patient in group $G1$ as in Program 5.4, in line 2. The assertion says that for each patient p in group $G1$, there must be the tuple, that consists of an atom Fin, in the relation $doActivity1$ at least one tuple.

For the patients in group $G2$, there are two requirements. The first requirement is that each patient must be done all activities in a given time. The another one is that each patient must start moving back to room $R2$ from the bath room in 5 minutes (inclusive) after finish taking a bath. From our specification of state transitions, any patient q in group $G2$ has already done all activities, if the patient q is in the $Fin2$ state. Moreover, any patient q start moving back in 5 minutes after finish taking a bath, if the patient q is in the $WaitForWalkBackR2$ state at most 5 minutes. We specify the requirement of patient in group $G2$ as in Program 5.4, from line 3 to line 4. The assertion says that:

· for each patient q in group $G2$, there must be the tuple, that consists of an atom $Fin2$, in the relation $doActivity2$ at least one tuple, and

· for each patient q in group $G2$, the number of tuples, that consists of an atom $WaitForWalkBackR2$, in the relation $doActivity2$ must be less than 6 tuples.

Program 5.4: Assertion that represents the requirements of hospital problem

```
1 assert Requirements {
2  all p: Patient1 | Fin in p.doActivity1.Time
3  all q: Patient2 | (Fin2 in q.doActivity2.Time) and
4                    (#(WaitForWalkBackR2.(q.doActivity2)) < 6)
5 }
```

Setting Scope and Checking the Assertion

To solve the problem, we use the *check* command for finding the counterexample of the assertion *Requirements*. Before executing the command, we need to set the scope of each signature. Program 5.5 represents the example of using *check* command and setting scope for solving hospital problem.

The scope of each signature should be set as following:

· **Patient1** : since there are exactly 3 patients in group $G1$, we need to set the scope of signature $Patient1$ with *exactly* 3.

· **Patient2** : since there are exactly 3 patients in group $G2$, we need to set the scope of signature $Patient2$ with *exactly* 3.

· **Time** : since the requirements say that all patients should be done all activities in 180 minutes, we should set the scope of signature $Time$ with *exactly* 180. However, we set the scope of $Time$ with *exactly* 182 because in our specification, we have two additional states, *Remain* state (or $Remain2$ state) and Fin state (or $Fin2$ state).

· **Nurse** : the scope of signature $Nurse$ represents the number of nurses in the hospital problem. To solve the problem, we need to find the sufficient scope of $Nurse$.

Since our specification of state transitions is the non-deterministic state transitions (from one current state, the patient can change to the two or more next states), we need to consider in the all possible situation of the system. By the way, Alloy Analyzer can generate all possible situations automatically. So that, to solve the problem (finding the number of nurses that is sufficient for the requirements), we just set the scope of signature $Nurse$ and execute *check* command. If there is a counterexample, it means that the number of nurses is not sufficient. So, we need to change the scope of $Nurse$ from the scope *exactly* 2 until we find the scope that the tool cannot find any counterexample. If we find the smallest number of nurses that there is no counterexample, it means that the number is the sufficient number of nurses.

Program 5.5: Example of Checking the Assertion and Setting Scope

```
1 check Requirements for exactly 3 Patient1,
2                        exactly 3 Patient2,
3                        exactly 7 Nurse,
4                        exactly 182 Time
```

5.3 Specification and Solving Problem in Maude

5.3.1 Specification in Maude

To specify the hospital problem in Maude, we specify the whole system with state transition system that states are expressed as soups, and the state transitions are specified with rewrite rules.

In the hospital problem, there is a limit of time for all patients to finish all activities. So, in each period of time (one minute), each patient can do only one activity (can move to only one state). Figure 5.2 represents the whole picture of state transitions of hospital problem in Maude. From the figure, there are two steps for the system to execute in one period of time. The first step is the *Checking Change* step. In this step, we set that will each patient changes states (or changes activities)? For the second step, it is the *Doing Activity* step. In this step, we let each patient to do one activity. Depending on the results from the *Checking Change* step, if any patient p does not want to change states, the patient p will be in the same state. Otherwise, he will change to the new states depends on the available resources. After done two steps, the system will decrease the minute by one for counting the period of time. If the minute equals zero, it means that the time is up, and the system must go to the *Finish* step.

For the state transition of hospital problem, there are one main state transition, and two sub-state transitions that represent the *Checking Change* step and *Doing Activity* step, respectively.

Specification of States

The states are expressed as a soup of observable components. Since there are three state transitions, there are three types of states for each state transition. The three types of states are:

- **State of main state transition** : there are 9 observable components that are described as following:
  - `(pc: l)` represents the current step of the system in each period of time, where l is a label of current step (the corresponding sort is `Label`).
  - `(DoAct[p][a]:n)` represents the number of minutes that each patient has already taken to do the current activity, where p is a patient identifier (the corresponding sort is `Pat`), a is an activity identifier (the corresponding sort is `Act`), and n is a natural number of minutes that has already used.

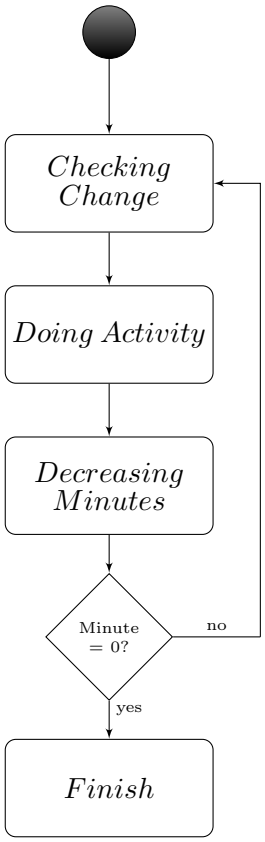

Figure 5.2: Diagram represents the whole picture of state transition in Maude

- `(Time: t)` represents the current period of time, where t is a natural number.
- `(Nurse: n)` represents the number of available nurses in the current period of time, where n is a natural number.
- `(RR: n)` represents the number of free spaces in the rehabilitation room in the current period of time, where n is a natural number.
- `(BR: n)` represents the number of free spaces in the bath room in the current period of time, where n is a natural number.
- `(change[p]: b)` represents the requirements of each patient that he want or does not want to change the state of activities, where p is a patient identifier, and b is a Boolean value. If b is *true*, the patient p wants to change the state. Otherwise, he does not want to change.
- `(waitingTime[p]: n)` represents the number of minutes that each patient in group $G2$ has used to wait after finish taking a bath, where p is a patient identifier, and n is a natural number.
- `(allPatients: s)` represents the set of patient identifiers of each patient, where s is a set that is expressed as a soup (the corresponding sort is `PatSet`).

- **State of sub-state transition of Checking Change step** : there are 6 observable components. The observable components are `(DoActC[p][a]: n)`, `(changeC[p]: b)`, `(NurseC: n)`, `(RRC: n)` and `(BRC: n)`, which represents the same things as `(DoAct[p][a]: n)`, `(change[p]: b)`, `(Nurse: n)`, `(RR: n)` and `(BR: n)`, respectively. However, for this type of state, the period of time is irrelevant. Moreover, there is another observable component that is `(remainPatientC: s)`. It represents the set of patients who is not set the requirement of changing state yet.

- **State of sub-state transition of Doing Activity step** : there are 7 observable components. The observable components are `(DoActD[p][a]: n)`, `(changeD[p]: b)`, `(NurseD: n)`, `(RRD: n)`, `(BRD: n)` and `(waitingTimeD: n)`, which represents the same things as `(DoAct[p][a]: n)`, `(change[p]: b)`, `(Nurse: n)`, `(RR: n)`, `(BR: n)` and `(waitingTime: n)`, respectively. However, for this type of state, the period of time is irrelevant. Moreover, there is another observable component that is `(remainPatientD: s)`. It represents the set of patients who have not done one activity yet.

Specification of State Transitions

We use rewrite rules to specify the state transition in Maude. As we described in the previous section, there are one main state transition and two sub-state transitions. Note that since the state transitions of step of doing activities of patient in group $G1$ and $G2$ are the same, we'll refer just to the specification of state transitions of patients in group $G1$. The description of each state transition are described as following:

Program 5.6: Example of Specification of Main State Transitions

```
1  crl[change] : (pc: change) (allPatient: PS) CONFIG1
2                      => (pc: do) (allPatient: PS) CONFIG2
3                  if (remainPatientC: PS) CONFIG1
4                      => (remainPatientC: patEmpty) CONFIG2 .
5
6  crl[do] : (pc: do) (Time: N) (allPatient: PS) CONFIG2
7                      => (pc: check) (Time: sd(N,1))
8                           (allPatient: PS) CONFIG3
9                  if (remainPatientD: PS) CONFIG2
10                     => (remainPatientD: patEmpty) CONFIG3 .
11
12 rl[check] : (pc: check) (Time: N)
13              => (pc: (if (N == 0) then finish else change fi))
14                 (Time: N) .
```

· **Main state transitions** : As we described in Figure 5.2, for each period of time, the system must execute two steps. First, the system must check whether does each patient want to change state by sending values of some observable components to the sub-state transitions of *Checking Change*. The sub-state transitions will execute and return one possible result from the given initial values. For the second step, the system must let each patient moves to the next states depends on the result values from *Checking Change* step. To do the second step, the system sends values of some observable components to the sub-state transitions of *Doing Activity*. The sub-state transition will execute and return one possible result to the main state transitions. After that, the system will decrease the value of observable component `(Time)` by 1. If the value of `(Time)` equals 0, the system will move to the *Finish* step. Otherwise, it will go back to the *Checking Change* step and execute the state transitions again.

To specify the main state transitions in Maude, we specify as in Program 5.6. In the Program 5.6, there are three rewrite rules, which each one is represented by different labels of `(pc)`.

· **In case of (pc: change)** : represents the system in the *Checking Change* step, where $CONFIG1$ is the values of other observable components. The system can change to the *do* state, and the values $CONFIG1$ can change to $CONFIG2$, if there are the values $CONFIG2$ that are the results of sub-state transition of *Checking Change* step, which execute with the initial values $CONFIG1$, and execute from the set *remainPatientC* consists of all patients until the set *remainPatientC* is empty.

· **In case of (pc: do)** : represents the system in the *Doing Activity* step, where $CONFIG2$ is the values of other observable components. The system can

change to the *check* state, the value of $Time$ can be decreased by 1, and the values $CONFIG2$ can change to $CONFIG3$, if there are the values $CONFIG3$ that are the results of sub-state transition of *Doing Activity* step, which execute with the initial values $CONFIG2$, and execute from the set $remainPatientD$ consists of all patients until the set $remainPatientD$ is empty.

- **In case of (pc: check)** : in this state, the system checks the value of $Time$. If the value of $Time$ equals 0, the system will go to the $finish$ state. Otherwise, it will go back to the *change* state and execute the state transitions again.

- **State transition of Checking Change step** : Figure 5.3 (a) represents the state transition of *Checking Change* step. This type of state transition is executed one time for each period of time by using the initial values that it receives from main state transition. In each execution, it checks whether will each patient want to change state by depending on the current state of activities of each patient, and the constraints of time to do each activity. After any patient p was checked, the patient p must be removed from the set $remainPatientC$.

  Since the rewrite rules of state transitions for each state of activities are the same, we'll show just the rewrite rules of $Meal$ state and $WaitForWalkToRR$ state.

  The rewrite rules of $Meal$ state are described as in Program 5.7, from line 1 to line 18. For any patient p that the current state is $Meal$ state, there are three conditions for patient p to change or do not change the state. The conditions are:

  - if any patient p has already taken time less than 30 minutes to have a meal, the patient p must not change the state (the value of `changeC[p]` must be $false$).
  - if any patient p has already taken time more than or equals 60 minutes to have a meal, the patient p must change the state (the value of `changeC[p]` must be $true$).
  - if any patient p has already taken time between 30 minutes to 60 minutes to have a meal, the patient p may or may not want to change the state (the value of `changeC[p]` is $true$ or $false$).

  Note that if any patient p want to change states, the patient p must release all the acquired resources of the current state before changing the states. For the case of current state is $Meal$ state, the patient p must release a resource nurse before changing the states by adding the number of available nurses with 1.

  The rewrite rule of $WaitForWalkToRR$ state as is described in Program 5.7, from line 20 to line 23. For any patient p that the current state is $WaitForWalkToRR$ state, the patient p must always try to change the state (the value of `changeC[p]` must always be $true$).

Program 5.7: Some Parts of Specification of Sub-State Transitions Checking Change

```
1  crl[mealC1]: (remainPatientC:(P PS))(DoActC[P][meal]: N1)(changeC[P]: B)
```

```
2                => (remainPatientC: PS)(DoActC[P][meal]:N1)
3                   (changeC[P]:false)
4                if (N1 < 30) .
5  crl[mealC2]: (remainPatientC:(P PS)) (DoActC[P][meal]: N1)
6                (changeC[P]: B) (NurseC: NN)
7                  => (remainPatientC: PS) (DoActC[P][meal]: N1)
8                     (changeC[P]: true) (NurseC: (NN + 1))
9                if (N1 >= 60) .
10 crl[mealC3]: (remainPatientC:(P PS)) (DoActC[P][meal]: N1)
11               (changeC[P]: B) (NurseC: NN)
12                 => (remainPatientC: PS) (DoActC[P][meal]: N1)
13                    (changeC[P]: true) (NurseC: (NN + 1))
14               if (N1 >= 30 and N1 < 60) .
15 crl[mealC4]: (remainPatientC:(P PS)) (DoActC[P][meal]: N1) (changeC[P]: B)
16                 => (remainPatientC: PS) (DoActC[P][meal]: N1)
17                    (changeC[P]: false)
18               if (N1 >= 30 and N1 < 60) .
19
20 rl[waitForWalkToRRC]: (remainPatientC: (P PS)) (DoActC[P][waitWalkRR]: N1)
21                       (changeC[P]: B)
22                         => (remainPatientC: PS) (DoActC[P][waitWalkRR]: N1)
23                            (changeC[P]: true) .
```

· **State transition of Doing Activity step** : Figure 5.3 (b) represents the state transition of *Doing Activity* step. Similar with the *Checking Change* step, it is executed one time for each period of time by using the initial values that it receives from main state transition. In each execution, it assigns the next states for each patient by depending on the result from *Checking Change* step, and the available resources. After any patient p was assigned the next state, the patient p must be removed from the set $remainPatientD$.

Since the rewrite rules of state transitions for each state of activities are the same (just names of states and conditions are difference), we'll show just the rewrite rules of $Meal$ state, $WaitForWalkToRR$ state and $WaitForWalkBackR2$ state.

The rewrite rules of $Meal$ state are described as in Program 5.8, from line 1 to line 12. For any patient p, in the set $remainPatientD$, that the current state is $Meal$ state, if he does not want to change the state (the value of `changeD[p]` is $false$), the next state of the patient p must be the same state ($Meal$ state). However, if the patient p wants to change the state (the value of `changeD[p]` is $true$), the next state is depending on the number of available nurses. If the number of available nurses is more than or equal 1, the next state must be $walkToRR$ state. Otherwise, the next state must be $waitForWalkToRR$ state.

The rewrite rule of $WaitForWalkToRR$ state is described as in Program 5.8, from line 14 to line 22. For any patient p, in the set $remainPatientD$, that the current state is $WaitForWalkToRR$ state, he must always want to change the state (the value of `changeD[p]` must always $true$). So, the next state of patient p is depending on the number of available nurses only. If the number of available nurses is more

than or equal 1, the next state must be $walkToRR$ state. Otherwise, the next state must be the same state ($waitForWalkToRR$ state).

The rewrite rule of $WaitForWalkBackR2$ state is described as in Program 5.8, from line 24 to line 33. This rewrite rule is the same as the rewrite rule of $WaitForWalkToRR$ state. However, in this rewrite rule, we must consider the value of observable component `(waitingTimeD[p])`. If any patient p can change the state to $WalkBackR2$ state, the system must record the time that patient p has taken to stay in the $WaitForWalkBackR2$ state in the component `(waitingTimeD[p])`.

Note that when any patient p can move to the new states, the patient p must acquire the available resources that he needs before changing states.

Program 5.8: Some Parts of Specification of Sub-State Transitions Doing Activity

```
1  crl[meal1]: (DoActD[P][meal]: N1) (changeD[P]: false) (usedPatientD: PS)
2                => (DoActD[P][meal]: (N1 + 1)) (changeD[P]: false)
3                   (usedPatientD: delPat(P,PS))
4              if (havePat(P,PS)) .
5  crl[meal2]: (DoActD[P][meal]: N1) (changeD[P]: true) (NurseD: NN)
6              (usedPatientD: PS)
7                => (DoActD[P][(if (NN >= 1) then walkToRR
8                                            else waitForWalkToRR fi)]: 1)
9                   (changeR[P]: false)
10                  (NurseR: (if (NN >= 1) then sd(NN,1) else NN fi))
11                  (usedPatientD: delPat(P,PS))
12             if (havePat(P,PS)) .
13
14  crl[waitWalkRR]: (DoActD[P][waitForWalkToRR]: N1) (changeD[P]: true)
15                   (NurseD: NN) (usedPatientD: PS)
16                    => (DoActD[P][(if (NN >= 1) then walkToRR
17                                                else waitForWalkToRR fi)]:
18                                     (if (NN >= 1) then 1 else (N1 + 1) fi))
19                       (changeD[P]: true)
20                       (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
21                       (usedPatientD: delPat(P,PS))
22                    if (havePat(P,PS)) .
23
24 crl[waitBackR2]: (DoActD[P][waitForWalkBackR2]: N1) (changeD[P]: true)
25                  (NurseD: NN) (waitingTimeD[P]: WT) (usedPatientD: PS)
26                   => (DoActD[P][(if (NN >= 2) then walkBackR2
27                                               else waitForWalkBackR2 fi)]:
28                                     (if (NN >= 2) then 1 else (N1 + 1) fi))
29                      (changeD[P]: true)
30                      (NurseD: (if (NN >= 2) then sd(NN,2) else NN fi))
31                      (waitingTimeD[P]: (if (NN >= 2) then N1 else WT fi))
32                      (usedPatientD: delPat(P,PS))
33                   if (havePat(P,PS)) .
```

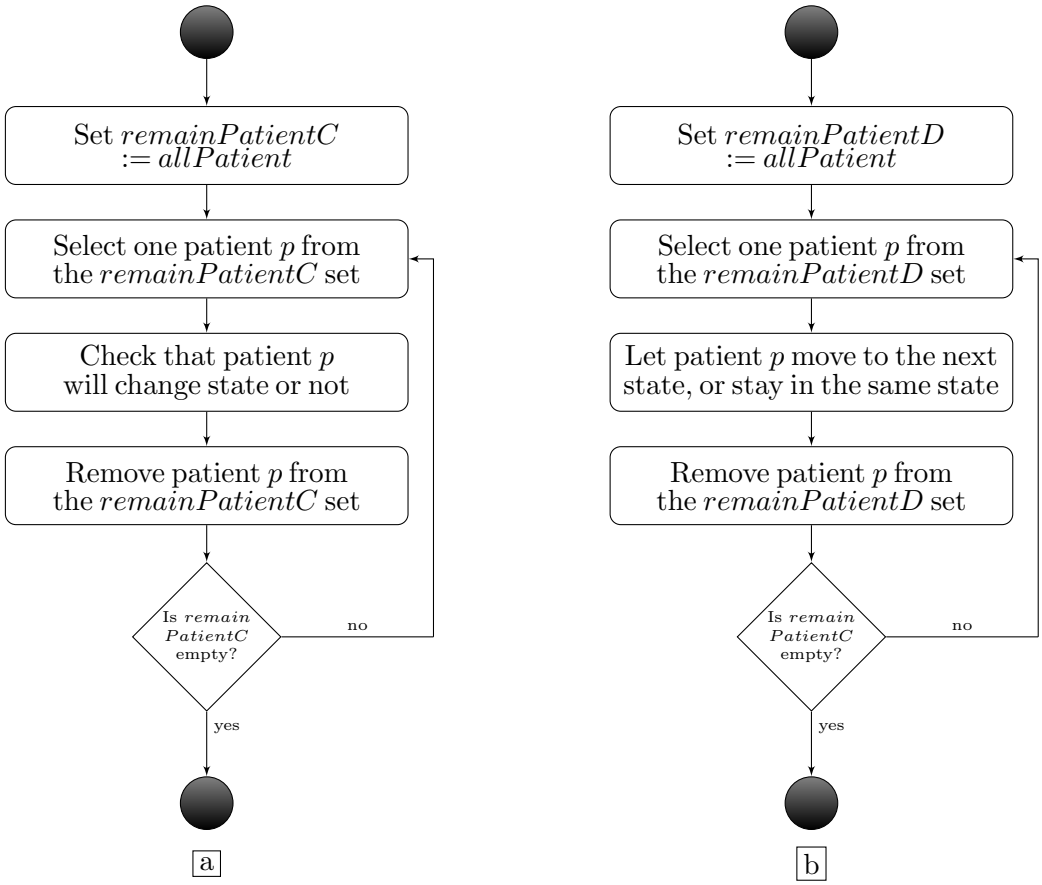

Figure 5.3: Diagram represents the state transition of *Checking Change* step (a), and the state transition of *Doing Activity* step (b)

5.3.2 Solving Problem in Maude

To solve the hospital problem in Maude, we consider the observable components when the system is in the $finish$ state. If the observable components hold the requirements, the system will move to the *correct* state. Otherwise, it will move to the $fail$ state. Moreover, we use the *search* command to search from an initial state to the state that the system is in the $fail$ state. If the *search* command cannot find an instance, it means that the number of nurses is sufficient for the requirements.

Specifying the rewrite rule of Finish State

Program 5.9 represents the rewrite rule of $finish$ state in the main state transition. When the system is in the $finish$ state, it will check the values of some observable components that are they hold the requirements? There are two requirements of the hospital problem. The requirements are:

· **All patients must done all activities in a given time** : to ensure this requirement,

when the system is in the $finish$ state, all patient in group $G1$ and $G2$ must be in the Fin and $Fin2$ state, respectively.

· **All patients must not wait after taking a bath more than 5 minutes** : to ensure this requirements, the values of observable components `waitingTime(p)`, for each patient p in group $G2$, must be less than 6.

If the observable components in the $finish$ state hold both conditions, it means that the current situation guarantees the requirements, and the system must move to the *correct* state. Otherwise, it must move to the $fail$ state.

Program 5.9: Specification of rewrite rule of finish state

```
1  rl[finish]: (pc: finish) (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1)
2              (DoAct[p[2]][A2]: N2) (DoAct[p[3]][A3]: N3)
3              (DoAct[p[4]][A4]: N4) (DoAct[p[5]][A5]: N5)
4              (waitingTime[p[3]]: WT3) (waitingTime[p[4]]: WT4)
5              (waitingTime[p[5]]: WT5)
6                => (pc: (if ((A0 == fin and A1 == fin and A2 == fin and
7                              A3 == fin2 and A4 == fin2 and A5 == fin2)
8                              and ((WT3 < 6) and (WT4 < 6) and (WT5 < 6)))
9                       then correct else fail fi))
10                     (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1)
11                     (DoAct[p[0]][A2]: N2) (DoAct[p[0]][A3]: N3)
12                     (DoAct[p[0]][A4]: N4) (DoAct[p[0]][A5]: N5)
13                     (waitingTime[p[3]]: WT3) (waitingTime[p[4]]: WT4)
14                     (waitingTime[p[5]]: WT5) .
```

Assigning the Initial State

Before executing the *search* command, we need to specify the values of observable components in the initial state. Program 5.10 represents one example of assigning the values of observable components in the initial state.

The values of each observable component should be as following:

· `(pc)`: the initial state of the system must be the *change* state.

· `(Time)`: same as solving problem in Alloy, we need to set the limit of time to be 182 periods of time (two more minutes for staying in *Remain* state and Fin state).

· `(DoAct[p])`: the initial state of activity of each patient in group $G1$ and $G2$ must be *Remain* and *Remain*2 state, respectively. Moreover, each patient must have already taken 1 minute. Note that for our specification, we represent the patients in group $G1$ by using patient identifiers `p[0], p[1]` and `p[2]`, and represent the patients in group $G2$ by using patient identifiers `p[3], p[4]` and `p[5]`.

· `(Nurse)`: the initial value of `Nurse` represents the number of nurses in the system. To solve the problem, we need to find the sufficient initial value of `Nurse`.

· `(change[p])`: the initial values of the requirement to change states of each patient p should be *true* because at the beginning, they must always try to change to the $Meal$ or $Meal2$ state.

· `(waitingTime[p])`: the initial values of the waiting time after finish taking a bath of each patient p in group $G1$ must be 0.

· `(allPatient)`: the initial value of set of all patients must be a set of all patient identifiers.

Note that we let `init` be the initial state.

Program 5.10: Example of values of observable components in the initial state

```
1 (Time: 182) (pc: change)
2 (DoAct[p[0]][rs]: 1) (DoAct[p[1]][rs]: 1) (DoAct[p[2]][rs]: 1)
3 (DoAct[p[3]][rs2]: 1) (DoAct[p[4]][rs2]: 1) (DoAct[p[5]][rs2]: 1)
4 (Nurse: 7) (RR: 2) (BR: 2)
5 (change[p[0]]: true) (change[p[1]]: true) (change[p[2]]: true)
6 (change[p[3]]: true) (change[p[4]]: true) (change[p[5]]: true)
7 (waitingTime[p[3]]: 0) (waitingTime[p[4]]: 0) (waitingTime[p[5]]: 0)
8 (allPatient: (p[0] p[1] p[2] p[3] p[4] p[5]))
```

Searching for Counterexamples

We use the *search* command to search in our state transitions for finding the counterexamples.

Since in each period of time, the different orders of patients to do one activity may generate the different situations of the system, we need to search in all possible orders of patients in each period. For example, in the system at the minute t, there are one patient p in group $G1$ that is in the $Meal$ state, one patient q in group $G2$ that is in the $InBR$ state, and the number of available nurses is 2. Assume that both patients want to change to the new state. If the system let the patient p change to the new state first, the situation at the next minute t' will be that the patient p is in the $WalkToRR$ state and the patient q is in the $WaitForWalkBackR2$ state. However, if the system let the patient q change to the new state first, the situation at the next minute t' will be that the patient p is in the $WaitForWalkToRR$ state and the patient q is in the $WalkBackR2$ state. So that, the different orders of patient to change states may generate different situations as in an example.

Furthermore, our specification of state transitions has the non-deterministic rewrite rules. The non-deterministic rules are in the case that when each patient may or may not want to change to the new states. So that, we need to search in all possible situations of requirements to change the state too.

By the way, the *search* command in Maude can search for finding counterexamples in all possible situations automatically. So that, we just assign the initial value of observable component `Nurse` and execute *search* command. We execute the *search* command as in Program 5.11. The *search* command will search from the given initial state to find

counterexamples, which are the situations that the system is in the $fail$ state. If there are counterexamples, it means that the number of nurses is not sufficient. So, we need to change the initial value of `Nurse` from 2 nurses until we find the value that the *search* command cannot find the counterexamples. If we find the smallest number of nurses that there is no counterexample, it means that the number is the sufficient number of nurses.

Program 5.11: Execution of Search command to find counterexamples

```
1 search in EXPERIMENT : init =>* (pc: fail) S:State .
```

5.3.3 Using Real-Time Maude

Besides specifying the hospital problem in Standard Maude, we also have specified it in Real-Time Maude.

Specification in Real-Time Maude

The specification of hospital problem in Real-Time Maude is almost similar with specification in Standard Maude. There are just some differences in both specifications. The differences are:

- In the state of main state transition, there is no the observable component `(Time)` because we can use the tick rule and timed search command to count the period of time.
- In the main state transition, there are only 2 state of the system, *change* and *do* state. We do not need the *check* state and $finish$ state anymore, since we can use the tick rule and timed search command to count the period of time.
- We change the rewrite rule of *do* state of the system to be the tick rule of 1 Time for counting the period of time after each patient has done one activity. For the other rewrite rules, we specified them to be the instantaneous rewrite rules.
- Since Real-Time Maude is based on Full Maude language, there are some different syntax that we need to change in our specification such as parenthesis.

Solving Problem in Real-Time Maude

To solve the hospital problem in Real-Time Maude, we do the same tasks as in Maude but we instead use the timed search command. To search, we assign the values of observable components in the initial state, which same as searching in Standard Maude (but there is no the observable component `(Time)`).

Program 5.12 represents the execution of timed search command for finding counterexamples from the given initial state. The timed search command will find the states of counterexamples. The states of counterexamples are the states that there are some patients in group $G2$, who have waited after finish taking a bath more than or equal 6

minutes, or there are some patients, who do not done all activities in the 182 periods of time. If it can find such state, it will show the counterexamples, and it means that the number of nurses is not sufficient. Same as in Standard Maude, we need to assign the value of `Nurse` from 2 nurses until the timed search command cannot find the counterexamples.

Program 5.12: Execution of Timed Search command to find counterexamples

```
(tsearch in EXPERIMENT: { init }
            =>* {(DoAct(p(0),A0:Act,N0:Nat)) (DoAct(p(1),A1:Act,N1:Nat))
                 (DoAct(p(2),A2:Act,N2:Nat)) (DoAct(p(3),A3:Act,N3:Nat))
                 (DoAct(p(4),A4:Act,N4:Nat)) (DoAct(p(5),A5:Act,N5:Nat))
                 (waitingTime(p(3),WT3:Nat)) (waitingTime(p(4),WT4:Nat))
                 (waitingTime(p(5),WT5:Nat)) S:State}
            such that (not(A0:Act == fin) or not(A1:Act == fin) or
                       not(A2:Act == fin) or not(A3:Act == fin2) or
                       not(A4:Act == fin2) or not(A5:Act == fin2) or
                       (WT3 >= 6) or (WT4 >= 6) or (WT5 >= 6))
      in time-interval between >= 182 and < 183 .)
```

Chapter 6

Results and Comparisons

6.1 Results from Alloy

When we do the experiments by executing the *check* command from the scope *exactly* 2 $Nurse$ to *exactly* 7 $Nurse$, the results are that Alloy Analyzer always find a counterexample. The counterexample, which Alloy Analyzer finds for each number of nurses, shows the path of situations from 1^{st} minute to 182^{nd} minute.

In this section, we will just show some parts of path of the counterexample for the case of 7 nurses. Before showing the counterexample, we will assign the names to each patient for easier to describe. Let the three patients in group $G1$ have names p_1, p_2 and p_3, and let the three patients in group $G2$ have names q_1, q_2 and q_3.

The parts of path of the counterexample is the path from 46^{th} minute to 53^{rd} minute, which represents a situation as follows:

· **In the** 46^{th} **minute**

- the patient p_1 is in the $Meal$ state, and acquires 1 nurse.
- the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
- the patient p_3 is in the $Meal$ state, and acquires 1 nurse.
- the patient q_1 is in the $Meal2$ state.
- the patient q_2 is in the $InBR$ state, and acquires 1 nurse.
- the patient q_3 is in the $InBR$ state, and acquires 1 nurse.
- there are 2 available nurses, 2 free spaces in RR, and no free space in BR

· **In the** 47^{th} **minute**

- the patient p_1 is in the $Meal$ state, and acquires 1 nurse.
- the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
- the patient p_3 is in the $Meal$ state, and acquires 1 nurse.

- the patient q_1 wants to change state and he can change. So the patient q_1 is in the $WalkToBR$ state, and acquires 2 nurses.
- the patient q_2 wants to change state but he cannot change because the number of available nurses is not sufficient. So the patient q_2 is in the $WaitForWalkBackR2$ state.
- the patient q_3 is in the $InBR$ state, and acquires 1 nurse.
- there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the** 48^{th} **minute**

  - the patient p_1 wants to change state and he can change. So the patient p_1 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
  - the patient p_3 is in the $Meal$ state, and acquires 1 nurse.
  - the patient q_1 is in the $WalkToBR$ state, and acquires 2 nurses.
  - the patient q_2 is also in the $WaitForWalkBackR2$ state.
  - the patient q_3 is in the $InBR$ state, and acquires 1 nurse.
  - there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the** 49^{th} **minute**

  - the patient p_1 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
  - the patient p_3 wants to change state and he can change. So the patient p_3 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient q_1 is in the $WalkToBR$ state, and acquires 2 nurses.
  - the patient q_2 is also in the $WaitForWalkBackR2$ state.
  - the patient q_3 is in the $InBR$ state, and acquires 1 nurse.
  - there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the** 50^{th} **minute**

  - the patient p_1 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
  - the patient p_3 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient q_1 is in the $WalkToBR$ state, and acquires 2 nurses.
  - the patient q_2 is also in the $WaitForWalkBackR2$ state.
  - the patient q_3 is in the $InBR$ state, and acquires 1 nurse.

- there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the 51^{st} minute**

  - the patient p_1 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_2 is in the $Meal$ state, and acquires 1 nurse.
  - the patient p_3 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient q_1 is in the $WalkToBR$ state, and acquires 2 nurses.
  - the patient q_2 is also in the $WaitForWalkBackR2$ state.
  - the patient q_3 is in the $InBR$ state, and acquires 1 nurse.
  - there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the 52^{nd} minute**

  - the patient p_1 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_2 wants to change state and he can change. So the patient p_2 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_3 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient q_1 wants to change state and he can change. So the patient q_1 is in the $InBR$ state, releases 2 nurses, and acquires 1 nurse.
  - the patient q_2 is also in the $WaitForWalkBackR2$ state.
  - the patient q_3 wants to change state and he can change. So the patient q_3 is in the $WalkBackR2$ state, releases 1 nurse, and acquires 2 nurses.
  - there are 1 available nurse, 2 free spaces in RR, and 1 free space in BR

- **In the 53^{rd} minute**

  - the patient p_1 wants to change state and he can change. So the patient p_1 is in the $InRR$ state, and releases 1 nurse.
  - the patient p_2 is in the $WalkToRR$ state, and acquires 1 nurse.
  - the patient p_3 wants to change state and he can change. So the patient p_3 is in the $InRR$ state, and releases 1 nurse.
  - the patient q_1 in the $InBR$ state, and acquires 1 nurse.
  - the patient q_2 can change state. So the patient q_2 is in the $WalkBackR2$ state, and acquires 2 nurses.
  - the patient q_3 has already done all activities.
  - there are 3 available nurses, no free space in RR, and 1 free space in BR

Table 6.1: The results of execution in Alloy with different number of nurses

Number of Nurses	**Time (mins)**	**Counterexample**
2	10.11	Yes
3	12.23	Yes
4	13.62	Yes
5	20.78	Yes
6	24.44	Yes
7	45.39	Yes
8	51.54	No
9	53.11	No
10	58.09	No

From some parts of path of counterexample, it shows that the patient q_2 must wait 6 minutes (from the 47^{th} minute to the 52^{nd} minute) after finish taking a bath (stay in the $WaitForWalkBackR2$ state 6 periods of time), which against the requirement. It means that the seven nurses are not sufficient for the system to hold the requirements.

However, when we execute the *check* command with the scope *exactly* 8 $Nurse$, Alloy Analyzer cannot find any counterexample. Moreover, same as the case of 8 nurses, the case of 9 and 10 nurses also do not have any counterexample. It means that by using Alloy, the sufficient number of nurses of hospital problem is 8 nurses.

Table 6.1 shows the results and time of executing the *check* command in Alloy by setting the different scope of $Nurse$. From the table, it shows that when executing with the scope of 2 nurses to 7 nurses, Alloy Analyzer can find the counterexample. However, from the scope of 8 nurses, there is no counterexample. Moreover, when the number of nurses is increasing, the time to execute is increasing. Because in the case of more number of nurses, it must use larger state spaces for finding the counterexample, so that takes more time to search. In addition, in the cases that do not have any counterexample, it use more time to execute than the cases that have counterexamples. Because when the tool found a counterexample, it stops searching and shows the counterexample immediately. So that, it searches in smaller state spaces than the cases that do not have counterexample, which the tool must search in overall state spaces.

6.2 Results from Maude

When we do the experiments by executing the *search* command, Maude cannot show any result, whenever in the case that just has only 2 nurses. Actually, Maude has taken time more than 10 days to execute, which is a too long time, but it does not return any result yet. So that, we cannot solve the hospital problem in the reasonable time by using Maude. However, it doesn't mean that Maude cannot solve the problem. If we let Maude to take more time to execute, it is possible that Maude can show the results, or may be it will execute until exceeding the limit of stack memory.

Moreover, we do more experiments by using the timed search command in Real-Time Maude. However, Real-Time Maude also cannot show any result because it exceeds the limit of stack memory. There are two possible reasons why it exceeds the limit of memory but Maude does not. The first reason is that Real-Time Maude may use more memory for executing timed search command than Standard Maude uses for executing *search* command. For the second reason, it is possible that Real-Time Maude can explore in the state spaces by timed search command faster than Maude. By the way, we can conclude that Real-Time Maude cannot be used to solve the hospital problem.

Besides using Maude to solve the hospital problem, we use it to recheck the results from Alloy. Since the counterexample of Alloy shows the path of situations from 1^{st} minute to 180^{th} minute, we can use Maude to explore that the path of counterexample from Alloy can be generated by using specification of Maude. So, the both languages (Standard Maude and Real-Time Maude) can guarantee that the counterexamples of Alloy, which have the number of nurses from 2 nurses to 7 nurses, can be generated by the models of Standard Maude and Real-Time Maude. However, in the case that there is no counterexample, we cannot use Maude to recheck the results from Alloy.

6.3 Comparisons of Alloy and Maude

From the experiment, which we specify the hospital problem in Alloy and Maude, we found differences between the both specification languages in many topics. The topics of differences are as following:

Specification techniques

Alloy: the specification technique of Alloy is based on relational logic. So, all structures of models of system that are specified in Alloy must be built from only atoms and relations. Moreover, the constraints of the system must be specified by using only the first-order logic. By using relational logic for specification, users must have deeply understanding about the system that they want to specify, because it is difficult to transform from the designed abstract model of system to be the model that based on atoms and relations. However, if the users can successfully specify the model of system, the model will small, simple, and can capture the underlying idea naturally.

Maude: the specification technique of Maude is based on rewriting logic. So, the models of system must be built from the rewrite rules. The rewrite rules are the relations between any two states, which are the sets of observable components. To specify the observable components, users can create any type of structures by defining the basic data structures such as list, set, tree, and tuples. By using rewriting logic, the users can straightforwardly specify the model from the designed abstract model. However, it is possible that the model of system becomes large and complex, which can decrease the efficiency of analyzer.

The techniques to handle with state transition system

Alloy: since all structures of models in Alloy can have only atoms and relations, Alloy cannot specify the state transition system directly. To specify the state transitions, users must create one set of atoms that represents the label of each step of state in state transition. So, this set must be one of the elements in tuples in all relations for telling the values of components in each step of state transitions. In the specification of hospital problem, for example, the signature $Time$ is such the set. Moreover, the users must use *predicate* and $fact$ paragraphs to specify the transitions between states by using the constraint implication. By using this technique, to specify the non-deterministic state transitions, the users just use the constraints implication and disjunction, which are easy to read and understand for other users.

Maude: since the specification technique of Maude is based on rewriting logic, the users can straightforwardly specify the state transitions by using the rewrite rules. Each rewrite rule can represent one transition from the current state to only one next state. So, to specify the non-deterministic state transitions, the users must use many rewrite rules that have the same current state but have different next states, which may be difficult to read and understand for other users.

The ways of executing the commands for analyzing

Alloy: to analyze the properties in Alloy, users must use the *assertion* paragraph for defining the properties, and use the *check* command for execute the assertion. Before executing the *check* command, the users must specify the scope of number of atoms in each set.

By the way, in the case of analyzing state transition system, the users whether or not specify the initial state of state transitions. If the users specify the initial state, Alloy Analyzer will start analyzing from the given initial state. Otherwise, it will generate all possible initial states automatically and starts analyzing from all initial states. In addition, the users must specify the scope of number of transitions in state transitions. From the *small scope hypothesis*, which says that if we examine all small cases, we are likely to find a counterexample, the users must select the appropriate scope that covers all small cases. If the scope is too small, the analyzing will not cover all reachable states. However, if the scope is too large, the analyzing will take too much time for executing. So, the appropriate scope is a scope that large enough to cover all reachable states and makes analyzing terminates quickly.

Maude: to analyze the properties in Maude, users can use the *search* command and the LTL model checker. However, in this research, we consider only the *search* command. Before executing the *search* command, the users must specify the initial state by assigning values of all observable components in the state. The *search* command will start exploring from the given initial state. However, the users do not need to specify the scope of number of transitions in state transitions because the *search* command will explore in all possible reachable states automatically. By the way, if the users want to bound the search, they can set the maximum depth d of searching to let the search terminates when it reaches

the depth d.

The ways of showing results

Alloy: after executing the *check* command, if Alloy Analyzer finds counterexamples, it will show only one possible instance from all counterexamples. The instance that the tool shows is a small case of counterexample, which users can understand and find the flaws of the specifications easily. In the case of analyzing state transition system, since all values in each step of states when applying transition rules in state transition (that we call *history*) are recorded in the relations, the counterexample can show the path of applying transition rules from the initial state to the end of scope. By showing the path, the users can understand that how the counterexample might arise easily. Moreover, the users can easily use the records of history to specify the constraints of the system. In addition, the results from Alloy Analyzer can be shown in a variety form, textual and graphical, that makes the users easily understand.

Maude: after executing the *search* command, if Maude finds counterexamples, it will show all possible instances of counterexamples. By showing all counterexamples, the users can see overall problems of specification easily, but the search must always explore in all reachable states. So, it is possible that the search does not terminate in the case of the reachable state space from the given initial state is infinite. Even if the users can bound the maximum number n of solutions to let the search terminates when it finds n counterexamples, but if there is no counterexample, the search also does not terminate. In addition, each counterexample, that is showed, shows only the last state of each solution. By the way, the users can use the `set traces on` command to let Maude shows each step of applying rewrite rules. However, since the `set traces on` command does not record the history, so the users cannot use the history directly. The results from Maude can be shown in only textual form, which is difficult for the users to understand.

Analysis techniques

Alloy: the analysis technique of Alloy Analyzer is based on SAT (boolean satisfiability) solver, which is one of the most efficient and widely used techniques. To analyze system, Alloy Analyzer translates constraints of the system into boolean constraints by using the negation of the assertion's constraints, and then solving with the SAT solver technique. If the SAT solver finds an assignment that makes the boolean constraints become *true*, such assignment is the counterexample. Otherwise, there is no counterexample. In the case of analyzing state transition system, Alloy Analyzer just translates all states and state transitions into boolean constraints, and use the SAT solver for finding the counterexample.

Maude: the analysis technique of the *search* command in Maude is based on breath-first search strategy. To analyze state transition system, the *search* command explores the counterexamples from the given initial state to all reachable states by applying all rewrite rules step by step. So, if the system is large and has many rewrite rules, it easily occurs the state explosion problem.

The appropriate specification languages for each type of systems

· **The concurrent system that is not too large** : Maude is more appropriate to specify than Alloy because the specification technique of Maude is based on rewriting logic, which is the best match for concurrent system. Moreover, by using the *search* command, the users do not need to bound the search and it can analyze in all possible reachable states automatically (assume that the whole state space is finite).

However, if the users specify such system in Alloy, the users cannot straightforwardly specify state transitions for the concurrent system. So, it is more difficult to specify than using Maude. Furthermore, the users need to set the appropriate scope of number of transitions. Normally, to know the appropriate scope, the users get from their own experience and do some experiments, which are difficult and cumbersome.

For example, the analysis of mutual exclusion property in a mutual exclusion protocol. If the users use Maude to analyze, they just define the initial state and execute the *search* command. And then the *search* command can automatically search in all reachable states. However, if the users use Alloy to analyze, at first, the user need to do some experiments for finding the appropriate scope of number of transitions, and then execute the *check* command to analyze in the scope.

By the way, in the case of the whole state space is infinite, to analyze in both specification languages, Alloy and Maude, the users need to specify the scope of analyzing.

· **The large and complex system** : even if the system is the concurrent system, Alloy is more appropriate to specify than Maude. Although, specifying the system in Alloy is more difficult than in Maude, but the model from Alloy is smaller and more simple than model from Maude because the model of Alloy is defined by using only simple relational logic.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

In this research, we have surveyed Alloy specification language, which is the new approach for software verification. We have described Alloy specification language with three key elements: a logic, a language, and an analysis. The specification technique of Alloy is based on relational logic, which combines the quantifiers of first-order logic with the operators of the relational calculus. The model that is specified by Alloy must consist of only atoms and relations. Moreover, there is Alloy Analyzer that is a tool for analyzing the properties in model of Alloy. The Alloy Analyzer is based on instance finding, which was inspired by SAT solver.

Furthermore, we have compared Alloy specification language with another specification language that is Maude specification language, which is an algebraic specification language. Maude specification language is one of the most famous and widely used specification languages. To compare between Alloy and Maude, we specify and analyze a non-trivial case study in both specification languages. The case study is the hospital problem, which were not specified by Alloy and Maude before. So, we have shown the approaches of specification and solving hospital problem in Alloy and Maude.

For the result of solving hospital problem, Alloy can solve the problem and the answer is that the sufficient number of nurses is 8 nurses, but Maude cannot solve the problem in the reasonable time. Besides using Maude to solve the problem, we use Maude to recheck the counterexamples from Alloy that are also generated by the model of Maude. So, the result is that the model of Maude can generate the situations of hospital problem that are the counterexample from Alloy. The reason why we use Maude to recheck the counterexamples is that because in Maude, we can straightforwardly specify the model from the designed abstract model. So, we can guarantee that if the counterexample can be generated from model of Maude, it must always occur in the real system.

In conclusion, we have used the experimental results to compare between Alloy and Maude. The experimental results that we use, are not just the result of solving hospital problem, but include many other things such as our experiences of using both specification languages to specify and solve the problem. So, we have analyzed all results and

shown the differences between Alloy and Maude in many topics, which we have described their advantages and disadvantages in each topic, such as the specification techniques, the analysis techniques, and the outputs from analysis. Even if Alloy can solve the hospital problem in the reasonable time but Maude cannot solve, it doesn't mean that Alloy is better than Maude because it depends on the nuances of each system. So, we have described the rough guidelines about the appropriate types of system for each specification language. However, the users should use our guidelines together with their prior experience to select the appropriate specification language for each system.

7.2 Future Works

7.2.1 Solving Hospital Problem in automatic approach

To solve hospital problem in both languages (Alloy and Maude), we need to find the sufficient number of nurses by specifying the number of nurses from 2 nurses until there is no any counterexample, which we manually do the experiment. So, one of our future works is that solving the hospital problem by automatic ways. We just ask the question to the system that how many nurses are sufficient, and then the system can automatically do the experiment and answers the sufficient number of nurses.

7.2.2 Comparing Alloy and Maude in other non-trivial case studies

from our experiment, The hospital problem can be solved by Alloy but cannot be solved by Maude in reasonable time. So, we should specify and analyze other nontrivial case studies in both languages. The other case studies should be the system that can be analyzed by Alloy and Maude in the reasonable time, which we can use the results to compare Alloy and Maude in more details. For example, the other non-trivial case studies are some consensus protocols that solve consensus problem in distributed system [Fis83], and some concurrency control protocols [SX08].

Appendix A

Specification of Hospital Problem in Alloy

```
module project/hospital
open util/ordering[Time]

sig Time {}
abstract sig Activity1 {}
one sig Remain extends Activity1 {}
one sig Meal extends Activity1 {}
one sig WaitForWalkToRR extends Activity1 {}
one sig WalkToRR extends Activity1 {}
one sig WaitForInRR extends Activity1 {}
one sig InRR extends Activity1 {}
one sig WaitForWalkBackR1 extends Activity1 {}
one sig WalkBackR1 extends Activity1 {}
one sig Fin extends Activity1 {}

abstract sig Activity2 {}
one sig Remain2 extends Activity2 {}
one sig Meal2 extends Activity2 {}
one sig WaitForWalkToBR extends Activity2 {}
one sig WalkToBR extends Activity2 {}
one sig WaitForInBR extends Activity2 {}
one sig InBR extends Activity2 {}
one sig WaitForWalkBackR2 extends Activity2 {}
one sig WalkBackR2 extends Activity2 {}
one sig Fin2 extends Activity2 {}

abstract sig Patient {}
sig Patient1 extends Patient {
 doActivity1: Activity1 one -> Time
}
sig Patient2 extends Patient {
 doActivity2: Activity2 one -> Time
}
one sig NoPatient extends Patient {}
```

```
sig Nurse {
 help: Patient one -> Time
}

pred init (t: Time) {
 all p: Patient1 | p.doActivity1.t = Remain
 all q: Patient2 | q.doActivity2.t = Remain2
}

pred notDo (t, t': Time, p: Patient1) {
 p.doActivity1.t' = Remain
}

pred haveMeal (t': Time, p: Patient1) {
 p.doActivity1.t' = Meal
}

pred waitForWalkRR (t': Time, p: Patient1) {
 p.doActivity1.t' = WaitForWalkToRR
}

pred walkRR (t': Time, p: Patient1) {
 p.doActivity1.t' = WalkToRR
}

pred waitForRR (t': Time, p: Patient1) {
 p.doActivity1.t' = WaitForInRR
}

pred RR (t': Time, p: Patient1) {
 p.doActivity1.t' = InRR
}

pred waitForWalkBack (t': Time, p: Patient1) {
 p.doActivity1.t' = WaitForWalkBackR1
}

pred walkBack (t': Time, p: Patient1) {
 p.doActivity1.t' = WalkBackR1
}

pred done (t': Time, p: Patient1) {
 p.doActivity1.t' = Fin
}

pred notDo2 (t': Time, p: Patient2) {
 p.doActivity2.t' = Remain2
}

pred haveMeal2 (t': Time, p: Patient2) {
 p.doActivity2.t' = Meal2
}
```

```
pred waitForWalkBR (t': Time, p: Patient2) {
 p.doActivity2.t' = WaitForWalkToBR
}

pred walkBR (t': Time, p: Patient2) {
 p.doActivity2.t' = WalkToBR
}

pred waitForBR (t': Time, p: Patient2) {
 p.doActivity2.t' = WaitForInBR
}

pred BR (t': Time, p: Patient2) {
 p.doActivity2.t' = InBR
}

pred waitForWalkBack2 (t': Time, p: Patient2) {
 p.doActivity2.t' = WaitForWalkBackR2
}

pred walkBack2 (t': Time, p: Patient2) {
 p.doActivity2.t' = WalkBackR2
}

pred done2 (t': Time, p: Patient2) {
 p.doActivity2.t' = Fin2
}

fact Traces {
 first.init
 all t: Time - last | let t' = t.next |
  all p: Patient1 |
   p.doActivity1.t = Remain implies
    (haveMeal[t',p] or notDo[t',p])
   else p.doActivity1.t = Meal implies
    (walkRR[t',p] or waitForWalkRR[t',p] or haveMeal[t',p])
   else p.doActivity1.t = WaitForWalkToRR implies
    (walkRR[t',p] or waitForWalkRR[t',p])
   else p.doActivity1.t = WalkToRR implies
    (RR[t',p] or waitForRR[t',p] or walkRR[t',p])
   else p.doActivity1.t = WaitForInRR implies
    (RR[t',p] or waitForRR[t',p])
   else p.doActivity1.t = InRR implies
    (walkBack[t',p] or waitForWalkBack[t',p] or RR[t',p])
   else p.doActivity1.t = WaitForWalkBackR1 implies
    (walkBack[t',p] or waitForWalkBack[t',p])
   else p.doActivity1.t = WalkBackR1 implies
    (done[t',p] or walkBack[t',p])
   else done[t',p]

 all t: Time - last | let t' = t.next |
```

```
  all q: Patient2 |
   q.doActivity2.t = Remain2 implies
    (haveMeal2[t',q])
   else q.doActivity2.t = Meal2 implies
    (walkBR[t',q] or waitForWalkBR[t',q] or haveMeal2[t',q])
   else q.doActivity2.t = WaitForWalkToBR implies
    (walkBR[t',q] or waitForWalkBR[t',q])
   else q.doActivity2.t = WalkToBR implies
    (BR[t',q] or waitForBR[t,t',q] or walkBR[t',q])
   else q.doActivity2.t = WaitForInBR implies
    (BR[t',q] or waitForBR[t',q])
   else q.doActivity2.t = InBR implies
    (walkBack2[t',q] or waitForWalkBack2[t',q] or BR[t',q])
   else q.doActivity2.t = WaitForWalkBackR2 implies
    (walkBack2[t,t',q] or waitForWalkBack2[t,t',q])
   else q.doActivity2.t = WalkBackR2 implies
    (done2[t,t',q] or walkBack2[t,t',q])
   else done2[t,t',q]
}

fact SetConstraints {
 all p: Patient1 | #(Meal.(p.doActivity1)) > 29 and
                   #(Meal.(p.doActivity1)) < 61
 all p: Patient1 | #(WalkToRR.(p.doActivity1)) > 2 and
                   #(WalkToRR.(p.doActivity1)) < 6
 all p: Patient1 | #(InRR.(p.doActivity1)) > 29 and
                   #(InRR.(p.doActivity1)) < 46
 all p: Patient1 | #(WalkBackR1.(p.doActivity1)) > 2 and
                   #(WalkBackR1.(p.doActivity1)) < 6
 all q: Patient2 | #(Meal2.(q.doActivity2)) > 29 and
                   #(Meal2.(q.doActivity2)) < 61
 all q: Patient2 | #(WalkToBR.(q.doActivity2)) > 0 and
                   #(WalkToBR.(q.doActivity2)) < 6
 all q: Patient2 | #(InBR.(q.doActivity2)) > 14 and
                   #(InBR.(q.doActivity2)) < 31
 all q: Patient2 | #(WalkBackR2.(q.doActivity2)) > 0 and
                   #(WalkBackR2.(q.doActivity2)) < 6

 all p: Patient1 | all t: Time |
         p.doActivity1.t = Remain implies #((help.t).p) = 0
 all p: Patient1 | all t: Time |
         p.doActivity1.t = Meal implies #((help.t).p) = 1
 all p: Patient1 | all t: Time |
         p.doActivity1.t = WaitForWalkToRR implies #((help.t).p) = 0
 all p: Patient1 | all t: Time |
         p.doActivity1.t = WalkToRR implies #((help.t).p) = 1
 all p: Patient1 | all t: Time |
         p.doActivity1.t = WaitForInRR implies #((help.t).p) = 0
 all p: Patient1 | all t: Time |
         p.doActivity1.t = InRR implies #((help.t).p) = 0
 all p: Patient1 | all t: Time |
         p.doActivity1.t = WaitForWalkBackR1 implies #((help.t).p) = 0
```

```
 all p: Patient1 | all t: Time |
        p.doActivity1.t = WalkBackR1 implies #((help.t).p) = 1
 all p: Patient1 | all t: Time |
        p.doActivity1.t = Fin implies #((help.t).p) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = Remain2 implies #((help.t).q) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = Meal2 implies #((help.t).q) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = WaitForWalkToBR implies #((help.t).q) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = WalkToBR implies #((help.t).q) = 2
 all q: Patient2 | all t: Time |
        q.doActivity2.t = WaitForInBR implies #((help.t).q) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = InBR implies #((help.t).q) = 1
 all q: Patient2 | all t: Time |
        q.doActivity2.t = WaitForWalkBackR2 implies #((help.t).q) = 0
 all q: Patient2 | all t: Time |
        q.doActivity2.t = WalkBackR2 implies #((help.t).q) = 2
 all q: Patient2 | all t: Time |
        q.doActivity2.t = Fin2 implies #((help.t).q) = 0

 all t: Time | #((doActivity1.t).InRR) < 3
   all t: Time | #((doActivity2.t).InBR) < 3

 all p: Patient1 | all t: Time - first |
        (p.doActivity1.t = Remain) implies  (#((help.t).NoPatient) < 1)
 all p: Patient1 | all t: Time - first |
        (p.doActivity1.t = WaitForWalkToRR) implies (#((help.t).NoPatient) < 1)
 all p: Patient1 | all t: Time - first |
        (p.doActivity1.t = WaitForInRR) implies (#((doActivity1.t).InRR) >= 2)
 all p: Patient1 | all t: Time - first |
        (p.doActivity1.t = WaitForWalkBackR1) implies (#((help.t).NoPatient) < 1)
 all q: Patient2 | all t: Time - first |
        (q.doActivity2.t = WaitForWalkToBR) implies (#((help.t).NoPatient) < 2)
 all q: Patient2 | all t: Time - first |
        (q.doActivity2.t = WaitForInBR) implies (#((help.t).NoPatient) < 1) or
                                                (#((doActivity2.t).InBR) >= 2)
 all q: Patient2 | all t: Time - first |
        (q.doActivity2.t = WaitForWalkBackR2) implies (#((help.t).NoPatient) < 2)
}

assert Requirements {
 all p: Patient1 | Fin in p.doActivity1.Time
 all q: Patient2 | (Fin2 in q.doActivity2.Time) and
                   (#(WaitForWalkBackR2.(q.doActivity2)) < 6)
}
```

Appendix B

Specification of Hospital Problem in Maude

```
fmod PATIENT is
 pr NAT .
 sort Pat .
 op p[_] : Nat -> Pat [ctor] .
 op equal : Pat Pat -> Bool .
 vars N1 N2 : Nat .

 eq equal(p[N1],p[N2]) = if (N1 == N2) then true else false fi .
endfm

fmod ACTIVITY is
 sort Act .
 ops rs meal waitWalkRR walkRR waitRR RR waitWalkBackR1 walkBackR1 fin : -> Act [ctor] .
 ops rs2 meal2 waitWalkBR walkBR waitBR BR waitWalkBackR2 walkBackR2 fin2 : -> Act [ctor] .
endfm

fmod SET{M :: TRIV} is
 pr NAT .
 sort Set .
 subsort M$Elt < Set .
 op empty : -> Set [ctor] .
 op _ _ : Set Set -> Set [ctor assoc comm id: empty] .
 op count : Set -> Nat .
 var N : M$Elt .
 var S : Set .
 eq N N = N .
endfm

view TrivToPatient from TRIV to PATIENT is
 sort Elt to Pat .
endv

fmod PATIENT-SET is
 pr SET{TrivToPatient} * (sort Set to PatSet, op empty : -> Set to patEmpty) .
 op havePat : Pat PatSet -> Bool .
```

```
 op delPat : Pat PatSet -> PatSet .

 vars P P1 : Pat .
 vars PS PS1 : PatSet .

 eq havePat(P,patEmpty) = false .
 eq havePat(P,(P1 PS)) = if equal(P,P1) then true else havePat(P,PS) fi .

 eq delPat(P,patEmpty) = patEmpty .
 eq delPat(P,(P1 PS)) = if equal(P,P1) then PS else (P1 delPat(P,PS)) fi .
endfm

fmod LABEL is
 sort Label .
 ops change do check finish correct fail : -> Label [ctor] .
endfm

fmod STATE-CHECK is
 pr BOOL .
 pr PATIENT .
 pr PATIENT-SET .
 pr ACTIVITY .
 sorts ObsC StateC .
 subsorts ObsC < StateC .

 op void : -> StateC [ctor] .
 op _ _ : StateC StateC -> StateC [ctor assoc comm id: void] .

 op (DoActC[_][_]:_) : Pat Act Nat -> ObsC [ctor] .
 op (remainPatientC:_) : PatSet -> ObsC [ctor] .
 op (changeC[_]:_) : Pat Bool -> ObsC [ctor] .
 op (NurseC:_) : Nat -> ObsC [ctor] .
 op (RRC:_) : Nat -> ObsC [ctor] .
 op (BRC:_) : Nat -> ObsC [ctor] .
endfm

fmod STATE-DO is
 pr BOOL .
 pr PATIENT .
 pr PATIENT-SET .
 pr ACTIVITY .
 pr LABEL .
 sorts ObsR StateR .
 subsorts ObsR < StateR .

 op void : -> StateR [ctor] .
 op _ _ : StateR StateR -> StateR [ctor assoc comm id: void] .

 op (DoActD[_][_]:_) : Pat Act Nat -> ObsR [ctor] .
 op (NurseD:_) : Nat -> ObsR [ctor] .
 op (remainPatientD:_) : PatSet -> ObsR [ctor] .
 op (changeD[_]:_) : Pat Bool -> ObsR [ctor] .
```

```
 op (RRD:_) : Nat -> ObsR [ctor] .
 op (BRD:_) : Nat -> ObsR [ctor] .
 op (waitingTimeD[_]:_) : Pat Nat -> ObsR [ctor] .
endfm

fmod STATE is
 pr BOOL .
 pr PATIENT .
 pr PATIENT-SET .
 pr ACTIVITY .
 pr LABEL .
 sorts Obs State .
 subsorts Obs < State .

 op void : -> State [ctor] .
 op _ _ : State State -> State [ctor assoc comm id: void] .

 op (pc:_) : Label -> Obs [ctor] .
 op (DoAct[_][_]:_) : Pat Act Nat -> Obs [ctor] .
 op (Time:_) : Nat -> Obs [ctor] .
 op (Nurse:_) : Nat -> Obs [ctor] .
 op (RR:_) : Nat -> Obs [ctor] .
 op (BR:_) : Nat -> Obs [ctor] .
 op (allPatient:_) : PatSet -> Obs [ctor] .
 op (change[_]:_) : Pat Bool -> Obs [ctor] .
 op (waitingTime[_]:_) : Pat Nat -> Obs [ctor] .
endfm

mod CHECK-CHANGE is
including STATE-CHECK .

vars P P1 : Pat .
vars PS PS1 : PatSet .
vars N N1 N2 NN NRR NBR : Nat .
var B : Bool .
vars A1 A2 : Act .

 rl[rsC1] : (remainPatientC: (P PS)) (DoActC[P][rs]: N1) (changeC[P]: B)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][rs]: N1) (changeC[P]: true) .

 crl[mealC1-1] : (remainPatientC: (P PS)) (DoActC[P][meal]: N1) (changeC[P]: B)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][meal]: N1) (changeC[P]: false)
         if (N1 < 30) .
 crl[mealC1-2] : (remainPatientC: (P PS)) (DoActC[P][meal]: N1) (changeC[P]: B)
                 (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][meal]: N1) (changeC[P]: true)
              (NurseC: (NN + 1))
         if (N1 >= 60) .
 crl[mealC1-3] : (remainPatientC: (P PS)) (DoActC[P][meal]: N1) (changeC[P]: B)
                 (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][meal]: N1)  (changeC[P]: true)
              (NurseC: (NN + 1))
```

```
        if (N1 >= 30 and N1 < 60) .
crl[mealC1-4] : (remainPatientC: (P PS)) (DoActC[P][meal]: N1)  (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][meal]: N1)  (changeC[P]: false)
        if (N1 >= 30 and N1 < 60) .

rl[waitWalkRRC1] : (remainPatientC: (P PS)) (DoActC[P][waitWalkRR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][waitWalkRR]: N1) (changeC[P]: true)  .

crl[walkRRC1-1] : (remainPatientC: (P PS)) (DoActC[P][walkRR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkRR]: N1) (changeC[P]: false)
        if (N1 < 3) .
crl[walkRRC1-2] : (remainPatientC: (P PS)) (DoActC[P][walkRR]: N1) (changeC[P]: B)
                  (NurseC: NN)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkRR]: N1) (changeC[P]: true)
             (NurseC: (NN + 1))
        if (N1 >= 5) .
crl[walkRRC1-3] : (remainPatientC: (P PS)) (DoActC[P][walkRR]: N1) (changeC[P]: B)
                  (NurseC: NN)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkRR]: N1) (changeC[P]: true)
             (NurseC: (NN + 1))
        if (N1 >= 3 and N1 < 5) .
crl[walkRRC1-4] : (remainPatientC: (P PS)) (DoActC[P][walkRR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkRR]: N1) (changeC[P]: false)
        if (N1 >= 3 and N1 < 5) .

rl[waitRRC1] : (remainPatientC: (P PS)) (DoActC[P][waitRR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][waitRR]: N1) (changeC[P]: true)  .

crl[RRC1-1] : (remainPatientC: (P PS)) (DoActC[P][RR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][RR]: N1)  (changeC[P]: false)
        if (N1 < 30) .
crl[RRC1-2] : (remainPatientC: (P PS)) (DoActC[P][RR]: N1) (changeC[P]: B)
               (RRC: NRR)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][RR]: N1) (changeC[P]: true)
             (RRC: (NRR + 1))
        if (N1 >= 45) .
crl[RRC1-3] : (remainPatientC: (P PS)) (DoActC[P][RR]: N1) (changeC[P]: B)
              (RRC: NRR)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][RR]: N1) (changeC[P]: true)
             (RRC: (NRR + 1))
        if (N1 >= 30 and N1 < 45) .
crl[RRC1-4] : (remainPatientC: (P PS)) (DoActC[P][RR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][RR]: N1) (changeC[P]: false)
        if (N1 >= 30 and N1 < 45) .

rl[waitWalkBackR1C1] : (remainPatientC: (P PS)) (DoActC[P][waitWalkBackR1]: N1)
                       (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][waitWalkBackR1]: N1)
             (changeC[P]: true) .

crl[walkBackR1C1-1] : (remainPatientC: (P PS)) (DoActC[P][walkBackR1]: N1)
                      (changeC[P]: B)
```

```
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR1]: N1)
             (changeC[P]: false)
        if (N1 < 3) .
crl[walkBackR1C1-2] : (remainPatientC: (P PS)) (DoActC[P][walkBackR1]: N1)
                      (changeC[P]: B) (NurseC: NN)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR1]: N1)
             (changeC[P]: true) (NurseC: (NN + 1))
        if (N1 >= 5) .
crl[walkBackR1C1-3] :  (remainPatientC: (P PS)) (DoActC[P][walkBackR1]: N1)
                      (changeC[P]: B) (NurseC: NN)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR1]: N1)
             (changeC[P]: true) (NurseC: (NN + 1))
        if (N1 >= 3 and N1 < 5) .
crl[walkBackR1C1-4] : (remainPatientC: (P PS)) (DoActC[P][walkBackR1]: N1)
                      (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR1]: N1)
             (changeC[P]: false)
        if (N1 >= 3 and N1 < 5) .

rl[finC1] : (remainPatientC: (P PS)) (DoActC[P][fin]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][fin]: N1) (changeC[P]: false) .

rl[rsC2] :  (remainPatientC: (P PS)) (DoActC[P][rs2]: N1) (changeC[P]: B)
          =>  (remainPatientC: delPat(P,PS)) (DoActC[P][rs2]: N1) (changeC[P]: true)  .

crl[mealC2-1] :  (remainPatientC: (P PS)) (DoActC[P][meal2]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][meal2]: N1)  (changeC[P]: false)
        if (N1 < 30) .
crl[mealC2-2] :  (remainPatientC: (P PS)) (DoActC[P][meal2]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][meal2]: N1) (changeC[P]: true)
        if (N1 >= 60) .
crl[mealC2-3] :  (remainPatientC: (P PS)) (DoActC[P][meal2]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][meal2]: N1) (changeC[P]: true)
        if (N1 >= 30 and N1 < 60) .
crl[mealC2-4] :  (remainPatientC: (P PS)) (DoActC[P][meal2]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][meal2]: N1) (changeC[P]: false)
        if (N1 >= 30 and N1 < 60) .

rl[waitWalkBRC2] : (remainPatientC: (P PS)) (DoActC[P][waitWalkBR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][waitWalkBR]: N1) (changeC[P]: true)  .

crl[walkBRC2-1] : (remainPatientC: (P PS)) (DoActC[P][walkBR]: N1) (changeC[P]: B)
          => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBR]: N1) (changeC[P]: false)
        if (N1 < 1) .
crl[walkBRC2-2] : (remainPatientC: (P PS)) (DoActC[P][walkBR]: N1) (changeC[P]: B)
                  (NurseC: NN)
          =>  (remainPatientC: delPat(P,PS)) (DoActC[P][walkBR]: N1) (changeC[P]: true)
             (NurseC: (NN + 2))
        if (N1 >= 5) .
crl[walkBRC2-3] : (remainPatientC: (P PS)) (DoActC[P][walkBR]: N1) (changeC[P]: B)
                  (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBR]: N1) (changeC[P]: true)
```

```
              (NurseC: (NN + 2))
          if (N1 >= 1 and N1 < 5) .
 crl[walkBRC2-4] : (remainPatientC: (P PS)) (DoActC[P][walkBR]: N1) (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][walkBR]: N1) (changeC[P]: false)
         if (N1 >= 1 and N1 < 5) .

 rl[waitBRC2] : (remainPatientC: (P PS)) (DoActC[P][waitBR]: N1) (changeC[P]: B)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][waitBR]: N1) (changeC[P]: true)  .

 crl[BRC2-1] : (remainPatientC: (P PS)) (DoActC[P][BR]: N1) (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][BR]: N1) (changeC[P]: false)
         if (N1 < 15) .
 crl[BRC2-2] : (remainPatientC: (P PS)) (DoActC[P][BR]: N1) (changeC[P]: B)
               (BRC: NBR) (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][BR]: N1) (changeC[P]: true)
              (BRC: (NBR + 1)) (NurseC: (NN + 1))
         if (N1 >= 30) .
 crl[BRC2-3] : (remainPatientC: (P PS)) (DoActC[P][BR]: N1) (changeC[P]: B)
               (BRC: NBR) (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][BR]: N1) (changeC[P]: true)
              (BRC: (NBR + 1)) (NurseC: (NN + 1))
         if (N1 >= 15 and N1 < 30) .
 crl[BRC2-4] :  (remainPatientC: (P PS)) (DoActC[P][BR]: N1) (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][BR]: N1) (changeC[P]: false)
         if (N1 >= 15 and N1 < 30) .

 rl[waitWalkBackR2C2] : (remainPatientC: (P PS)) (DoActC[P][waitWalkBackR2]: N1)
                        (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][waitWalkBackR2]: N1)
              (changeC[P]: true)  .

 crl[walkBackR2C2-1] : (remainPatientC: (P PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: false)
         if (N1 < 1) .
 crl[walkBackR2C2-2] : (remainPatientC: (P PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: B)
                       (NurseC: NN)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: true)
              (NurseC: (NN + 2))
         if (N1 >= 5) .
 crl[walkBackR2C2-3] : (remainPatientC: (P PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: B)
                       (NurseC: NN)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: true)
              (NurseC: (NN + 2))
         if (N1 >= 1 and N1 < 5) .
 crl[walkBackR2C2-4] : (remainPatientC: (P PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: B)
           =>  (remainPatientC: delPat(P,PS)) (DoActC[P][walkBackR2]: N1) (changeC[P]: false)
         if (N1 >= 1 and N1 < 5) .

 rl[finC2] : (remainPatientC: (P PS)) (DoActC[P][fin2]: N1) (changeC[P]: B)
           => (remainPatientC: delPat(P,PS)) (DoActC[P][fin2]: N1)  (changeC[P]: false) .
endm
```

```
mod DO is
 including STATE-DO .

 vars P P1 : Pat .
 vars PS PS1 : PatSet .
 vars N N1 N2 NN WT NRR NBR : Nat .
 var B : Bool .
 vars A1 A2 : Act .

 crl[rs1] : (DoActD[P][rs]: N1) (changeD[P]: true) (NurseD: NN) (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 1) then meal else rs fi)]:
                  (if (NN >= 1) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[meal1-1] : (DoActD[P][meal]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][meal]: (N1 + 1)) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
 crl[meal1-2] : (DoActD[P][meal]: N1) (changeD[P]: true) (NurseD: NN)
                (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 1) then walkRR else waitWalkRR fi)]: 1)
             (changeD[P]: false) (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
        if (havePat(P,PS)) .

 crl[waitWalkRR1] : (DoActD[P][waitWalkRR]: N1) (changeD[P]: true) (NurseD: NN)
                    (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 1) then walkRR else waitWalkRR fi)]:
                   (if (NN >= 1) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[walkRR1-1] : (DoActD[P][walkRR]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][walkRR]: (N1 + 1)) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
 crl[walkRR1-2] : (DoActD[P][walkRR]: N1) (changeD[P]: true) (RRD: NRR)
                  (remainPatientD: PS)
          => (DoActD[P][(if (NRR >= 1) then RR else waitRR fi)]: 1) (changeD[P]: false)
             (RRD: (if (NRR >= 1) then sd(NRR,1) else NRR fi)) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[waitRR1] : (DoActD[P][waitRR]: N1) (changeD[P]: true) (RRD: NRR) (remainPatientD: PS)
          =>  (DoActD[P][(if (NRR >= 1) then RR else waitRR fi)]:
                   (if (NRR >= 1) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (RRD: (if (NRR >= 1) then sd(NRR,1) else NRR fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[RR1-1] : (DoActD[P][RR]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][RR]: (N1 + 1)) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
```

```
crl[RR1-2] : (DoActD[P][RR]: N1) (changeD[P]: true) (NurseD: NN) (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 1) then walkBackR1 else waitWalkBackR1 fi)]: 1)
             (changeD[P]: false) (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[waitWalkBackR1] : (DoActD[P][waitWalkBackR1]: N1) (changeD[P]: true) (NurseD: NN)
                      (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 1) then walkBackR1 else waitWalkBackR1 fi)]:
                 (if (NN >= 1) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (NurseD: (if (NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[walkBackR1-1] : (DoActD[P][walkBackR1]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][walkBackR1]: (N1 + 1)) (changeD[P]: false)
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
crl[walkBackR1-2] : (DoActD[P][walkBackR1]: N1) (changeD[P]: true) (remainPatientD: PS)
          => (DoActD[P][fin]: 1) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[fin1] : (DoActD[P][fin]: N1) (remainPatientD: PS)
          => (DoActD[P][fin]: (N1 + 1)) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[rs2] : (DoActD[P][rs2]: N1) (changeD[P]: true) (remainPatientD: PS)
          => (DoActD[P][meal2]: 1) (changeD[P]: true) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[meal2-1] : (DoActD[P][meal2]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][meal2]: (N1 + 1)) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
crl[meal2-2] : (DoActD[P][meal2]: N1) (changeD[P]: true) (NurseD: NN) (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 2) then walkBR else waitWalkBR fi)]: 1)
             (changeD[P]: false) (NurseD: (if (NN >= 2) then sd(NN,2) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[waitWalkBR2] : (DoActD[P][waitWalkBR]: N1) (changeD[P]: true) (NurseD: NN)
                   (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 2) then walkBR else waitWalkBR fi)]:
                  (if (NN >= 2) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (NurseD: (if (NN >= 2) then sd(NN,2) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

crl[walkBR2-1] : (DoActD[P][walkBR]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][walkBR]: (N1 + 1)) (changeD[P]: false)
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
crl[walkBR2-2] : (DoActD[P][walkBR]: N1) (changeD[P]: true) (BRD: NBR) (NurseD: NN)
```

```
                (remainPatientD: PS)
          => (DoActD[P][(if (NBR >= 1 and NN >= 1) then BR else waitBR fi)]: 1)
             (changeD[P]: false)
             (BRD: (if (NBR >= 1 and NN >= 1) then sd(NBR,1) else NBR fi))
             (NurseD: (if (NBR >= 1 and NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[waitBR2] : (DoActD[P][waitBR]: N1) (changeD[P]: true) (BRD: NBR) (NurseD: NN)
                (remainPatientD: PS)
          => (DoActD[P][(if (NBR >= 1 and NN >= 1) then BR else waitBR fi)]:
                 (if (NBR >= 1 and NN >= 1) then 1 else (N1 + 1) fi))
             (changeD[P]: true)
             (BRD: (if (NBR >= 1 and NN >= 1) then sd(NBR,1) else NBR fi))
             (NurseD: (if (NBR >= 1 and NN >= 1) then sd(NN,1) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[BR2-1] : (DoActD[P][BR]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][BR]: (N1 + 1)) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
 crl[BR2-2] : (DoActD[P][BR]: N1) (changeD[P]: true) (NurseD: NN) (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 2) then walkBackR2 else waitWalkBackR2 fi)]: 1)
             (changeD[P]: false) (NurseD: (if (NN >= 2) then sd(NN,2) else NN fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[waitWalkBackR2] : (DoActD[P][waitWalkBackR2]: N1) (changeD[P]: true) (NurseD: NN)
                       (waitingTimeD[P]: WT) (remainPatientD: PS)
          => (DoActD[P][(if (NN >= 2) then walkBackR2 else waitWalkBackR2 fi)]:
                 (if (NN >= 2) then 1 else (N1 + 1) fi))
             (changeD[P]: true) (NurseD: (if (NN >= 2) then sd(NN,2) else NN fi))
             (waitingTimeD[P]: (if (NN >= 2) then N1 else WT fi))
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[walkBackR2-1] : (DoActD[P][walkBackR2]: N1) (changeD[P]: false) (remainPatientD: PS)
          => (DoActD[P][walkBackR2]: (N1 + 1)) (changeD[P]: false)
             (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
 crl[walkBackR2-2] : (DoActD[P][walkBackR2]: N1) (changeD[P]: true) (remainPatientD: PS)
          => (DoActD[P][fin2]: 1) (changeD[P]: false) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .

 crl[fin2] : (DoActD[P][fin2]: N1) (remainPatientD: PS)
          =>  (DoActD[P][fin2]: (N1 + 1)) (remainPatientD: delPat(P,PS))
        if (havePat(P,PS)) .
endm

mod EXPERIMENT is
 including STATE .
 including CHECK-CHANGE .
```

```
including DO .

vars P P1 : Pat .
vars PS PS1 : PatSet .
vars N N0 N1 N2 N3 N4 N5 N02 N12 N22 N32 N42 N52 NN NRR NBR NN2 NRR2 NBR2
     WT3 WT4 WT5 WT32 WT42 WT52 T : Nat .
vars A0 A1 A2 A3 A4 A5 A02 A12 A22 A32 A42 A52 : Act .
vars B B0 B1 B2 B3 B4 B5 B02 B12 B22 B32 B42 B52 : Bool .

crl[change] : (pc: change)
               (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1) (DoAct[p[2]][A2]: N2)
               (DoAct[p[3]][A3]: N3) (DoAct[p[4]][A4]: N4) (DoAct[p[5]][A5]: N5)
               (change[p[0]]: B0) (change[p[1]]: B1) (change[p[2]]: B2)
               (change[p[3]]: B3) (change[p[4]]: B4) (change[p[5]]: B5)
               (Nurse: NN) (RR: NRR) (BR: NBR) (allPatient: PS)
          => (pc: do)
             (DoAct[p[0]][A02]: N02) (DoAct[p[1]][A12]: N12) (DoAct[p[2]][A22]: N22)
             (DoAct[p[3]][A32]: N32) (DoAct[p[4]][A42]: N42) (DoAct[p[5]][A52]: N52)
             (change[p[0]]: B02) (change[p[1]]: B12) (change[p[2]]: B22)
             (change[p[3]]: B32) (change[p[4]]: B42) (change[p[5]]: B52)
             (Nurse: NN2) (RR: NRR2) (BR: NBR2) (allPatient: PS)
        if (remainPatientC: PS)
           (DoActC[p[0]][A0]: N0) (DoActC[p[1]][A1]: N1) (DoActC[p[2]][A2]: N2)
           (DoActC[p[3]][A3]: N3) (DoActC[p[4]][A4]: N4) (DoActC[p[5]][A5]: N5)
           (changeC[p[0]]: B0) (changeC[p[1]]: B1) (changeC[p[2]]: B2)
           (changeC[p[3]]: B3) (changeC[p[4]]: B4) (changeC[p[5]]: B5)
           (NurseC: NN) (RRC: NRR) (BRC: NBR)
             => (remainPatientC: patEmpty)
                (DoActC[p[0]][A02]: N02) (DoActC[p[1]][A12]: N12) (DoActC[p[2]][A22]: N22)
                (DoActC[p[3]][A32]: N32) (DoActC[p[4]][A42]: N42) (DoActC[p[5]][A52]: N52)
                (changeC[p[0]]: B02) (changeC[p[1]]: B12) (changeC[p[2]]: B22)
                (changeC[p[3]]: B32) (changeC[p[4]]: B42) (changeC[p[5]]: B52)
                (NurseC: NN2) (RRC: NRR2) (BRC: NBR2) .

crl[do] : (pc: do) (Time: T)
          (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1) (DoAct[p[2]][A2]: N2)
          (DoAct[p[3]][A3]: N3) (DoAct[p[4]][A4]: N4) (DoAct[p[5]][A5]: N5)
          (change[p[0]]: B0) (change[p[1]]: B1) (change[p[2]]: B2)
          (change[p[3]]: B3) (change[p[4]]: B4) (change[p[5]]: B5)
          (Nurse: NN) (RR: NRR) (BR: NBR)
          (waitingTime[p[3]]: WT3) (waitingTime[p[4]]: WT4) (waitingTime[p[5]]: WT5)
          (allPatient: PS)
          => (pc: check) (Time: (sd(T,1)))
             (DoAct[p[0]][A02]: N02) (DoAct[p[1]][A12]: N12) (DoAct[p[2]][A22]: N22)
             (DoAct[p[3]][A32]: N32) (DoAct[p[4]][A42]: N42) (DoAct[p[5]][A52]: N52)
             (change[p[0]]: B02) (change[p[1]]: B12) (change[p[2]]: B22)
             (change[p[3]]: B32) (change[p[4]]: B42) (change[p[5]]: B52)
             (Nurse: NN2) (RR: NRR2) (BR: NBR2)
             (waitingTime[p[3]]: WT32) (waitingTime[p[4]]: WT42) (waitingTime[p[5]]: WT52)
             (allPatient: PS)
        if (remainPatientD: PS)
           (DoActD[p[0]][A0]: N0) (DoActD[p[1]][A1]: N1) (DoActD[p[2]][A2]: N2)
```

```
            (DoActD[p[3]][A3]: N3) (DoActD[p[4]][A4]: N4) (DoActD[p[5]][A5]: N5)
            (changeD[p[0]]: B0) (changeD[p[1]]: B1) (changeD[p[2]]: B2)
            (changeD[p[3]]: B3) (changeD[p[4]]: B4) (changeD[p[5]]: B5)
            (NurseD: NN) (RRD: NRR) (BRD: NBR)
            (waitingTimeD[p[3]]: WT3) (waitingTimeD[p[4]]: WT4) (
            waitingTimeD[p[5]]: WT5)
              => (remainPatientD: patEmpty)
                 (DoActD[p[0]][A02]: N02) (DoActD[p[1]][A12]: N12) (DoActD[p[2]][A22]: N22)
                 (DoActD[p[3]][A32]: N32) (DoActD[p[4]][A42]: N42) (DoActD[p[5]][A52]: N52)
                 (changeD[p[0]]: B02) (changeD[p[1]]: B12) (changeD[p[2]]: B22)
                 (changeD[p[3]]: B32) (changeD[p[4]]: B42) (changeD[p[5]]: B52)
                 (NurseD: NN2) (RRD: NRR2) (BRD: NBR2)
                 (waitingTimeD[p[3]]: WT32) (waitingTimeD[p[4]]: WT42)
                 (waitingTimeD[p[5]]: WT52) .

 rl[check] : (pc: check) (Time: N)
           => (pc: (if (N == 0) then finish else change fi)) (Time: N) .

 rl[finish] : (pc: finish)
              (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1) (DoAct[p[2]][A2]: N2)
              (DoAct[p[3]][A3]: N3) (DoAct[p[4]][A4]: N4) (DoAct[p[5]][A5]: N5)
              (waitingTime[p[3]]: WT3) (waitingTime[p[4]]: WT4) (waitingTime[p[5]]: WT5)
           => (pc: (if ((A0 == fin and A1 == fin and A2 == fin and
                         A3 == fin2 and A4 == fin2 and A5 == fin2) and
                        ((WT3 < 6) and (WT4 < 6) and (WT5 < 6)))
                    then correct else fail fi))
              (DoAct[p[0]][A0]: N0) (DoAct[p[1]][A1]: N1) (DoAct[p[0]][A2]: N2)
              (DoAct[p[0]][A3]: N3) (DoAct[p[0]][A4]: N4) (DoAct[p[0]][A5]: N5)
              (waitingTime[p[3]]: WT3) (waitingTime[p[4]]: WT4) (waitingTime[p[5]]: WT5) .
endm
```

References

[Bac93] Jean Bacon. *Concurrent systems: an integrated approach to operating systems, database, and distributed systems*. Addison-Wesley, 1993.

[BBF+10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[CDE+07] Manuel Clavel, Francisco DurÃ¡n, Steven Eker, Patrick Lincoln, Narciso MartÃ-Oliet, JosÃ© Meseguer, and Carolyn Talcott. Ltl model checking. In *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*, pages 385–418. Springer Berlin Heidelberg, 2007.

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*, January 2011.

[DFO03] Razvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. Cafeobj: Logical foundations and methodologies. *Computing and Informatics*, 22, 2003.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[EM11] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[Fis83] MichaelJ. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In Marek Karpinski, editor, *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer Berlin Heidelberg, 1983.

[GCKP08] Paul S Grisham, Charles L. Chen, Sarfraz Khurshid, and Dewayne E. Perry. Validation of a security model with the alloy analyzer, 2008.

[GKK+88] Joseph Goguen, Claude Kirchner, Helene Kirchner, Aristide Megrelis, Jose Meseguer, and Timothy Winkler. An introduction to obj 3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer Berlin Heidelberg, 1988.

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*. 2008.

[Jac01] Daniel Jackson. Lightweight formal methods. In JosÃ©Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 2001.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. *SIGSOFT Softw. Eng. Notes*, 25(5):14–25, August 2000.

[Lam00] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

[LXLZ09] Guoqi Li, Yang Xiao, Minyan Lu, and Yuchao Zhang. A formal method based case study for access control. In *Computational Intelligence and Security, 2009. CIS '09. International Conference on*, volume 2, pages 460–463, Dec 2009.

[Mai83] David Maier. Relational operators. In *The Theory of Relational Databases*. Computer Science Press, 1983.

[ND90] J.P. Jouannaud N. Dershowitz. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. The MIT Press/Elsevier (1990), 1990.

[Olv07] P. C. Olveczky. *Real-Time Maude 2.3 Manual*, 2007.

[PPZ01] J. Meseguer C. Talcott P.C. POlveczky, M. Keaton and S. Zabele. Specification and analysis of the aer/nca active network protocol suite in real-time maude. In *4th International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–348. Springer-Verlag, 2001.

[RV07] Adrian Riesco and Alberto Verdejo. The eigrp protocol in maude. 2007.

[Smu68] Raymond M. Smullyan. *First-Order Logic*. Springer Berlin Heidelberg, 1968.

[SX08] Shashi Shekhar and Hui Xiong. Concurrency control protocols. In Shashi Shekhar and Hui Xiong, editors, *Encyclopedia of GIS*, pages 128–128. Springer US, 2008.

[SyF08] Anthony J. H. Simons and Carlos Alberto Fernandez y Fernandez. Using alloy to model-check visual design notations. *CoRR*, abs/0802.2258, 2008.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[zsp06] Z specification language. In *Mathematical Approaches to Software Quality*, pages 75–91. Springer London, 2006.