Toward a Model Checker for Ambient Logic Using the Process Analysis Toolkit

by

Yujie Sun

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Sherbrooke, Quebec, Canada
January 2015

Copyright © Yujie Sun, 2015

Abstract

Model checking has emerged as an effective verification method in both software and hardware development. A dedicated model checker is thus a useful tool for most application domains. Ambient Calculus is a process calculus where processes may reside within a hierarchy of locations. Processes can then move through the location hierarchy and modify it. The Ambient Logic is a modal logic designed to specify properties of distributed and mobile computations programmed in the Ambient Calculus. It includes logical operators that can be used to make assertions about locations and their names.

This thesis is concerned on how to build a model checker for mobile ambient processes against specification described as formulas in the Ambient Logic. For a given program written in ambient calculus and its properties specified as an ambient logic formula, the ambient model checker will be used to determine automatically whether the process satisfies the specification (logic formula). We start the work toward such an ambient model checker by implementing the syntax and semantics of the Ambient Calculus as a PAT (Process Analysis Toolkit) module. We also implemented in this module a syntactic definition for the Ambient Logic. The actual implementation of the model checking algorithm is the subject of future work.

Acknowledgments

First of all, I'd like to thank my supervisor, Dr. Stefan Bruda. During my work on this project, he gave me generous help. Without his help, I'm not able to complete this work.

I also want to thank Dr. Liu Yang and his team member, they developed the Process Analysis Toolkit. Thanks for their work and answers to my questions.

Table of Contents

List of Figures

Chapter 1 Introduction

Model checking is an automatic verification technique for both software and hardware systems. When developing a software or hardware computing system (such as program, protocol, or circuit diagram) we would like to ensure the correctness of our design. One way of establishing correctness is model checking [6], an automatic verification technique to formally ensure that a system observes a given specification. To use model checking we first need to construct a model for the system using a formal language such as a process algebra [14, 15]. We then express the desired properties of the system using either some temporal logic or even the same process algebraic language used to model the system. Afterward, the model checking algorithm will perform an exhaustive search of the state space of our system model to determine if the given property is true or not. Model checking is automated, efficient and reliable. The most serious disadvantage of model checking is the state explosion problem.

Modal logic is a type of formal logic that extends classical propositional or predicate logic to include operators expressing modality. For example, the statement "Anna is happy" might be qualified by saying that Anna is usually happy, in which case the term "usually" is functioning as a modal (or word that expresses modality). The traditional modalities of truth, include "possibility", "necessity" and "impossibility". In order to present and reason about the different propositions specifically and exactly, some other modalities has been formalized in modal logic, including temporal modalities (modalities of time), deontic modalities, epistemic modalities and so on [9]. Modal logic

represents modalities using modal operators. For example, "Anna may be happy tomorrow" and "It's possible that Anna will be happy tomorrow" both express the notion of "possibility". In modal logic, this could be represented as an operator “Possibly” followed by "Anna will be happy tomorrow".

Temporal logic is a variant of modal logic for expressing temporal modalities and so representing propositions qualified in terms of time. In the temporal logic we can express statements like "Anna is always happy", "Anna will eventually be happy", or "Anna will be happy until I leave". Same as the modal logic, in temporal logic we use modal operators to represent modalities. For example, the unary modal operators include ○ for "Next" (holds in the next step), ◇ for "Future" (holds in some place on the subsequent path of events), and □ for "Globally" (holds in the entire subsequent path or events).

Temporal logic has found an important application in formal verification in general, and model checking in particular. It is used to state the properties that our system model needs to satisfy. An example of such a property (expressible in temporal logic) is "whenever a request is made, access to a resource is eventually granted, but it is never granted to two requestors simultaneously".

Several temporal logics have been developed in the past, including but not limited to LTL and CTL [6]. Such (“classical”) temporal logic formalisms however are less suitable for constructions such as "somewhere there is a virus", or "eventually the agent crosses the firewall". In other words, it is difficult to express properties of mobile

computations using classical temporal logic. Ambient Calculus and the associated Ambient Logic [2, 3] have been proposed specifically to address this shortcoming. These formalisms are created o facilitate the specification and verification of mobile computational environments.

The Ambient Calculus is a framework for describing the mobility of both software and hardware. An ambient is a bounded place where computation happens. That is, an ambient is a named cluster of running processes and nested sub-ambient. Given this nesting (tree-like) spatial structure, each computation state could be expressed in a special ambient tree. The mobility is represented by the re-arrangement of the ambient tree whenever an ambient moves in to or out of another ambient. This is how the Ambient Calculus constructs the system model for mobile computations.

Ambient Logic is a variant of temporal logic. It contains not only the standard temporal modalities to describe the evolution of ambient processes, but also spatial modalities to describe the tree structure of ambient processes. Ambient logic could therefore specify properties of distributed and mobile computations programmed using the Ambient Calculus. Given a system model written in Ambient Calculus and properties specified using Ambient Logic, one would like to determine automatically whether the system model satisfies the specification through model checking. This thesis is concerned with the construction of a model checker for mobile ambient processes against specifications described as formulas in ambient logic, which we will call "ambient model checker". We complete only the first steps in this pursuit by offering an implementation of the syntax and semantics of the Ambient Calculus, as well as the syntax of Ambient

Logic.

1.1 Thesis Organization

The remainder of this thesis is organized as follows: in Chapter 2 we introduce some background knowledge, which includes the Ambient Calculus, Ambient Logic, as well as PAT3. Among them, the Process Analysis Toolkit [8, 11] is our development environment. The third version (PAT3) is a self-contained framework to support composing, simulating and reasoning about concurrent, real-time systems [10, 16]. It features a layered architecture design with an intermediate representation layer (IRL), so it can handle the modeling languages and model checking algorithms separately, and so the model checking algorithms can be shared by different languages. This kind of layered design make it possible to develop model checkers for a wide range of application domains. This is the reason for choosing PAT3 as our development environment. An abbreviated "user guide" is included at the end of Chapter 2, including the instructions for building the development environment and running the model checker.

In Chapter 3, we introduced the related previous work. In fact the only practical implementation of Ambient Calculus is the Ambient Designer built by Coronato and De Pietro [4], which will be introduced briefly in this chapter.

In order to be able to construct a model checker we had to restrict the Ambient Calculus. Indeed, in the process of settling the complexity bounds of the model checking problem for Ambient Calculus with public names against the Ambient Logic [7] it was discovered that the model checking problem will be undecidable if either the calculus

contains replication or the logic contains the guarantee operator. Therefore we removed the replication component from the calculus and so simplified the ambient calculus to avoid undecidable problems. A discussion on the matter can be found in Chapter 4.

After comparing LTL (Linear-time Temporal Logic) and the Ambient Logic, we found that the significant gap between these two modal logics is the concept of ambient modalities including locations, spatial constructs and so on. To some extent, we can extend LTL to get Ambient Logic by adding ambient modalities to LTL. So we could reuse part of the LTL model checking algorithms generated by the PAT3 module generator. In Chapter 5 we show the extension design and our partial implementation of the ambient model checker.

Chapter 2 Preliminaries

This chapter introduces the necessary background, including the Ambient Calculus, the Ambient Logic and the Process Analysis Toolkit.

2.1 Ambient Calculus

The Ambient Calculus is a process calculus, in which the processes could reside within a hierarchy of locations and even modify it. To make the basic Ambient Calculus [5] more powerful, Cardelli and Gordon introduced a modified version of Ambient Calculus [3], which strengthened the definition of structural congruence to express the intended equivalence of spatial configurations. In the paper we will construct a model checker for

this enriched version of Ambient Calculus. We need however to make some modification first, given the complexity and decidability of their algorithms. Figure 1 presents the syntax of processes [3].

Processes

P,Q,R :: =	processes
(vn)P	restriction
0	void
P | *Q*	composition
!P	replication
M[P]	ambient
M.P	capability
(*n*).*P*	input action
<*M*>	output action
M :: =	capabilities
n	name
in M	can enter into *M*
out M	can exit out of *M*
open M	can open *M*
ε	null
M.M'	path

Figure 1: Syntax of processes. The process constructs are separated into spatial and temporal. The first five are spatial process constructs, and the following three are temporal process constructs.

The process constructs are separated into spatial and temporal. In the above table, the first five constructs: restriction, void, composition, replication and ambient are spatial process constructs. (vn)P is used to describe the restricted name n in process P. The process 0 stands for "void", for example, n[0] means there is nothing at the place n[]. Similarly, M[P] means the process P resides at the place M[]. P | Q means the Process P and Process

Q exist in the same system at the same time. The following three constructs are temporal process constructs. Capability describe the movement of processes, which includes name, enter into, exit out of, open, null and path. Input action and Output action are both used to describe the information exchange.

Structural Congruence

$P \equiv P$	(Structural Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Structural Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Structural Trans)
$P \equiv Q \Rightarrow (vn)P \equiv (vn)Q$	(Structural Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Structural Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Structural Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Structural Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Structural Action)
$P \equiv Q \Rightarrow (n).P \equiv (n).Q$	(Structural Input)
$\varepsilon.P \equiv P$	(Structural ε)
$(M.M').P \equiv M.M'.P$	(Structural .)
$(vn)(vm)P \equiv (vm)(vn)P$	(Structural Res Res)
$(vn)\mathbf{0} \equiv \mathbf{0}$	(Structural Res Zero)
$(vn)(P\|Q) \equiv P\|(vn)Q$, if $n \notin fn(P)$	(Structural Res Par)
$(vn)m[P]) \equiv m[(vn)P]$, if $n \neq m$	(Structural Res Amb)
$P \mid \mathbf{0} \equiv P$	(Structural Par Zero)
$P \mid Q \equiv Q \mid P$	(Structural Par Comm)
$(P \mid Q)\mid R \equiv P \mid (Q \mid R)$	(Structural Par Assoc)
$!\mathbf{0} \equiv \mathbf{0}$	(Structural Repl Zero)
$!(P \mid Q) \equiv !P \mid !Q$	(Structural Repl Par)
$!P \equiv P \mid !P$	(Structural Repl Copy)
$!P \equiv !!P$	(Structural Repl Repl)

Figure 2: Structural congruence.

The following syntactic conventions are used:

- Parentheses are used for precedence.
- The process **0** stands for "void", so the contexts n[**0**] and M.**0** are the same as n[] and M, respectively.

- The composition operator has the lowest precedence. For example, the expression $M.P|Q$ is read $(M.P)|Q$.

To define reductions we need to explain the structural congruence relation between processes. Figure 2 shows the structural congruence rules [3], which are crucial for the definition of satisfaction (which will be introduced later).

To represent temporal modality, Cardelli and Gordon defined the reduction relation, shown in Figure 3 [3]. During the design of our ambient model checker, we will use the Ambient Calculus as described in this section, except for one small modification.

Reduction

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Reduction In)
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Reduction Out)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Reduction Open)
$P \equiv Q \Rightarrow (vn)P \equiv (vn)Q$	(Reduction Comm)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Structural Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Structural Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Structural Action)
$P \equiv Q \Rightarrow (n).P \equiv (n).Q$	(Structural Input)
$\mathcal{E}.P \equiv P$	(Structural ε)
$(M.M').P \equiv M.M'.P$	(Structural .)
$(vn)(vm)P \equiv (vm)(vn)P$	(Structural Res Res)
$(vn)\mathbf{0} \equiv \mathbf{0}$	(Structural Res
$(vn)(P \mid Q) \equiv P \mid (vn)Q$, if $n \notin fn(P)$	(Structural Res Par)
$(vn)m[P]) \equiv m[(vn)P]$, if $n \neq m$	(Structural Res

Figure 3: The reduction relation, describing the dynamic behavior of ambient.

2.2 Ambient Logic

In a modal logic the truth of a formula is relative to a state (or world). However, in ambient logic the truth of a space-time modal formula is relative to the here and now [3].

In Cardelli and Gordon's opinion, "each formula talks about the current time and the current place, which means, the current state of execution at the current location." For example, the formula n[0] is read: there is here and now an empty location called n. Universal and existential quantification is introduced on top of this concept. The syntax of Ambient Logical formulas is shown in Figure 4. The satisfaction relation denotes whether a processes satisfy a specific formula: $\mathcal{P} \vDash \mathcal{A}$ means the process $\mathcal{P}$ satisfies the closed formula $\mathcal{A}$. The definition of the satisfaction relation is shown in Figure 5.

Logic Formulas

η, μ	A name *n* or a variable *x*
$\mathcal{A}, \mathcal{B}, \mathcal{C} ::=$	
T	true
$\neg\ \mathcal{A}$	negation
$\mathcal{A} \vee \mathcal{B}$	disjunction
0	void
$\mathcal{A} \mid \mathcal{B}$	composition
$\mathcal{A}\ \rhd\ \mathcal{B}$	guarantee
$\eta\ [\mathcal{A}]$	location
$\mathcal{A}\ @\ \eta$	placement
$\eta\ ®\ \mathcal{A}$	revelation
$\mathcal{A}\ \oslash\ \eta$	hiding
$\Diamond\ \mathcal{A}$	sometime modality
$\blacklozenge\ \mathcal{A}$	somewhere modality
$\forall\ x.\mathcal{A}$	universal quantification

Figure 4: The syntax of ambient logic formulas. The first three formulas give classical propositional logic. The next five describe a tree-like structure of locations. The two spatial and temporal modalities make assertions about states that may happen "further away" in space or time, respectively.

Temporal modalities are represented by reduction in the operational semantics of the ambient calculus. The relation $P \downarrow P'$ indicates that P contains P' within one level of ambient nesting, and the relation $P \downarrow^* P'$ indicates that P contains P' at some nesting level. That is, the relation $\downarrow^*$ is the reflexive and transitive closure of the relation $\downarrow$.

Satisfaction

$\Pi ::=$	sort of processes
$\Phi ::=$	sort of formulas
$\vartheta ::=$	sort of variables
$\Lambda ::=$	sort of names

$\forall P \in \Pi.$	$P \models T$		
$\forall P \in \Pi, \mathcal{A} \in \Phi .$	$P \models \neg \mathcal{A}$	$\triangleq$	$\neg P \models \mathcal{A}$
$\forall P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi .$	$P \models \mathcal{A} \vee \mathcal{B}$	$\triangleq$	$P \models \mathcal{A} \vee P \models \mathcal{B}$
$\forall P \in \Pi.$	$P \models \mathbf{0}$	$\triangleq$	$P \equiv \mathbf{0}$
$\forall P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi .$	$P \models \mathcal{A} \mid \mathcal{B}$	$\triangleq$	$\exists P', P'' \in \Pi. P \equiv P' \mid P'' \wedge P' \models \mathcal{A} \wedge P'' \models \mathcal{B}$
$\forall P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi .$	$P \models \mathcal{A} \rhd \mathcal{B}$	$\triangleq$	$\forall P' \in \Pi. P' \models \mathcal{A} \Longrightarrow P' \mid P'' \models \mathcal{B}$
$\forall P \in \Pi, n \in \Lambda, \mathcal{A} \in \Phi .$	$P \models n[\mathcal{A}]$	$\triangleq$	$\exists P' \in \Pi. P \equiv n[P'] \wedge P' \models \mathcal{A}$
$\forall P \in \Pi, \mathcal{A} \in \Phi .$	$P \models \mathcal{A}@n$	$\triangleq$	$n[P] \models \mathcal{A}$
$\forall P \in \Pi, n \in \Lambda, \mathcal{A} \in \Phi.$	$P \models n \circledR \mathcal{A}$	$\triangleq$	$\exists P' \in \Pi. P \equiv (\upsilon n) P' \wedge P' \models \mathcal{A}$
$\forall P \in \Pi, \mathcal{A} \in \Phi .$	$P \models n \oslash \mathcal{A}$	$\triangleq$	$(\upsilon n) P \models \mathcal{A}$
$\forall P \in \Pi, \mathcal{A} \in \Phi .$	$P \models \Diamond \mathcal{A}$	$\triangleq$	$\exists P' \in \Pi. P \rightarrow^* P' \wedge P' \models \mathcal{A}$
$\forall P \in \Pi, \mathcal{A} \in \Phi .$	$P \models \blacklozenge \mathcal{A}$	$\triangleq$	$\exists P' \in \Pi. P \downarrow^* P' \wedge P' \models \mathcal{A}$
$\forall P \in \Pi, x \in \vartheta, \mathcal{A} \in \Phi .$	$P \models \forall x. \mathcal{A}$	$\triangleq$	$\forall m \in \Lambda. P \models \mathcal{A}\{x \leftarrow m\}$

Figure 5: The satisfaction relation of Ambient Logic.

The first three formulas (T, $\neg\mathcal{A}$, $\mathcal{A} \vee \mathcal{B}$)) give classical propositional logic. All processes satisfy the formula T. A process satisfies the formula $\neg\mathcal{A}$ if it does not satisfy the formula $\mathcal{A}$. A process satisfies the formula $\mathcal{A} \vee \mathcal{B}$ if it satisfies the formula $\mathcal{A}$ or the formula $\mathcal{B}$. The following formulas form the core of process logic. A process satisfies the formula **0**, only if it is a void process. A process P satisfies the formula $\mathcal{A} \mid \mathcal{B}$, if P consists of two composition processes, such as P' | P", and P' satisfies $\mathcal{A}$ while P" satisfies $\mathcal{B}$. A process satisfies the formula n [$\mathcal{A}$] if there exists a process P' such that P has the form n [P′] with P' satisfying $\mathcal{A}$. The connectives @ and ⊳ can be used to express security properties. A process P satisfies the formula $\mathcal{A}@n$ if P satisfies $\mathcal{A}$ even when placed into the location n. And a process P satisfies the formula $\mathcal{A} \rhd \mathcal{B}$ if P manages to satisfies $\mathcal{B}$ under any possible attack by an opponent that is bound to satisfy $\mathcal{A}$. The

connectives ® and ∅ are used to express properties about names. Specifically, ® is used to *reveal* a restricted name, and ∅ is used to *hide* (restrict) a name. A process P satisfies the formula $\Diamond \mathcal{A}$ if $\mathcal{A}$ holds in the future for some residual P' of P. And a process P satisfies the formula $\blacklozenge \mathcal{A}$ if $\mathcal{A}$ holds at some sublocation P' within P, where the sublocation is defined by $P \downarrow^{*} P'$.

During the designing of our ambient model checker we follow Ambient Logic as presented above, but excluding the guarantee operator. We will further make some small modifications which will be presented later.

2.3 Process Analysis Toolkit (PAT)

PAT is a self-contained, extensible and modularized, multi-domain model checking systems for composing, simulating and reasoning about concurrent, real-time, probabilistic systems, as well as other possible domains [10, 16].

The main reason we chose PAT as our development platform is that PAT3 adopts a layered design with an intermediate representation layer (IRL), which separates modeling languages from model checking algorithms so that the algorithms can be shared by different languages. In addition, the PAT3 layered architecture is specifically designed for extensibility, which provides different interfaces (APIs) for domain experts to create customized model checkers with minimum efforts. PAT3 could then be extended in several ways. One such a way (which we will adopt) is using the module generator tool to create a new PAT3 module to support a new language, which in our case is the ambient

model checker module.

2.4 Development Environment

Our development platform consists of PAT, Visual Studio 2010 Professional, and ANTLRWorks 1.4.2. The PAT module generator tool is used to generate the module project (in C#) with interface classes, language classes and code skeleton. To work with the generated code, we also need the Visual Studio environment, since the generated module is a complete C# solution with two projects. In addition, the parser classes are not automatically generated in the module project, so we need to create parser classes using Antlrworks. The development environment is configured as follows.

The latest version of PAT and the corresponding version of the .Net Framework are installed first. If using Linux, Unix, Mac OS, etc. the Mono package also needs to be installed, and a suitable version of PAT should be chosen. The installation instructions are available on the PAT Web site [11].

ANTLR [1] is a tool that creates a parser and a lexer in some target language (such as Java) based on a given grammars, as well as the necessary runtime. We use ANTLRWorks to run the ANTLR tool, and we also need the associated library. We therefore install ANTLRWorks and then follow the instructions to get the runtime environment to run the generated parsers/lexers. We also need the Java development environment. Indeed, even though we use the C# target of ANTLR 4, the Java environment is still required for compiling applications. A complete user guide for working with ANTLR 4 is available elsewhere [12, 13].

Chapter 3 Previous work

To the best of our knowledge there is only one practical implementation of Ambient Calculus, namely the Ambient Designer. We will describe this system below.

Antonio Coronato and Giuseppe De Pietro developed Ambient Designer to support graphical model and test specifications [4]. Figure 6 shows a screen shot of the GUI of their Ambient Designer, in which they describe one patient, Ellie, moving in a clinic. They set the patient and all the rooms in the clinic as classes of ambients.

Initially, at time T0, the patient is expressed as an ambient named Ellie, located in the Acceptance Room. In ambient Ellie there are 3 possible actions, which are the capabilities of this ambient: 0 (out of the acceptance room), 1 (in injection room), 2 (set is injected). A flag "isWaiting" is also set in this ambient to express the state.

Their Ambient Designer enables the step-by-step reduction of Ambient Calculus terms. The designer uses graphical objects and dialog boxes to specify the initial term and properties; the tool will then obtain the equivalent textual representation of the term. For example, while the ambient Ellie performs its three actions, the state T0 can evolve into T1, T2 and so on. This means that the ambient Ellie will move out of the acceptance room, move into the injection room and then get injected.

While reducing a term, the Ambient Designer validates the specified properties. For example, suppose that it is specified that the patient can only enter the injection room

when his status is "isWaiting". Then, if a patient with a different status tries to move in this room, an exception will be generated.

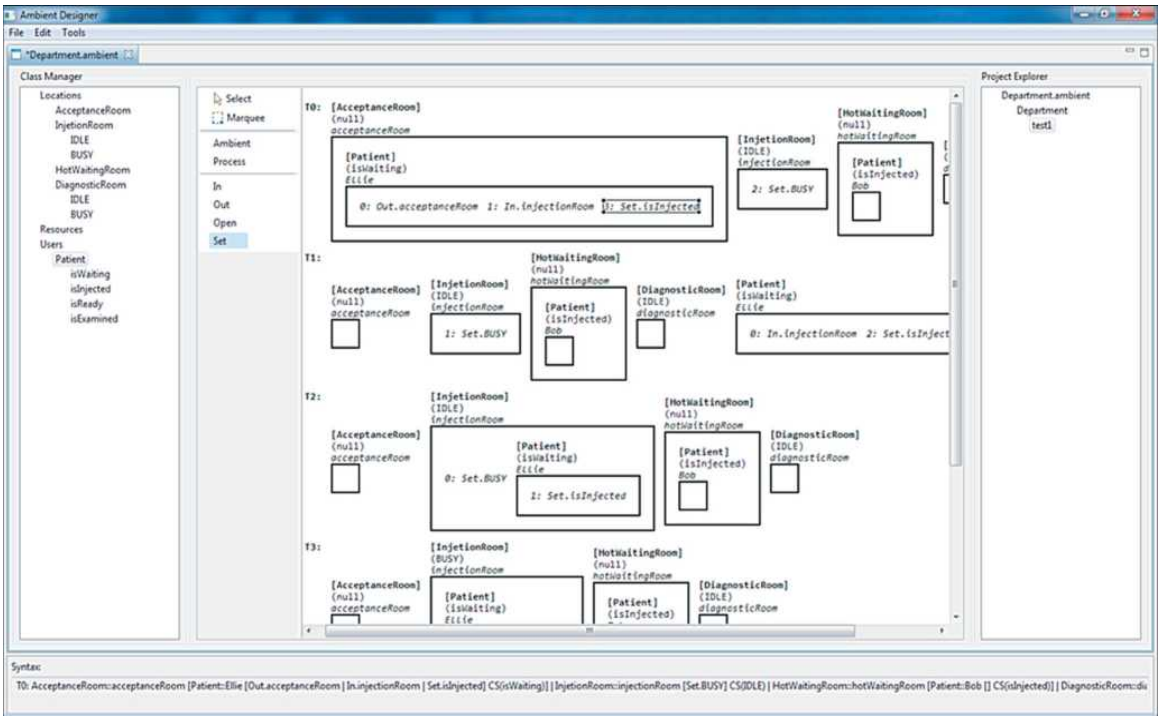


Figure 6: GUI of the Ambient Designer (Coronato and De Pietro).

Although the Ambient Designer supports testing specific execution paths, it cannot prove the correctness of the entire specification. In this thesis, we will start the design of a true model checker for Ambient Calculus against Ambient Logic. Our model checker reads in the textual representation (rather than the graphical representation used in the Ambient Designer) of the system model written in Ambient Calculus, and the specified properties expressed by Ambient Logic. Then, it will eventually validate the specified properties formally. By contrast the Ambient Designer does not offer formal proofs of correctness.

In addition, we borrow the concept of “Event” from CSP and introduce in in the Ambient Calculus to express the state changes of ambients. The following chapter will explain this in more details.

Chapter 4 Design Considerations

Our development platform PAT3 adopts a layered design, so we will perform our design and development in three layers (Modeling layer, Intermediate Representation Layer, and Analysis Layer).

4.1 Modeling Layer

The modeling layer of PAT is the supported application domain, such as distributed systems, service oriented computing, security protocol, etc. In our case (Ambient Calculus and Logic) the modeling layer identifies the syntax of the language. The main components in the modeling layer are the language parser and the model components, including syntax classes, variables, channels, and so on. In other words, when we provide a textual model, the parser will check the correctness of the syntax and generate the model components. Next, we will create the language parser for Ambient Logic.

Recall that Ambient Logic talks about properties of mobile computations, such as "the agent has gone away", "eventually the agent crosses the firewall", "every agent always carries a suitcase", or "there is always at most one agent called n here". All of these are properties that hold at particular locations. Therefore we need to design a parser

which can recognize not only processes but also the locations. This can be accomplished along two lines of thought that will be illustrated using following example.

4.2 Example Problem Description

Nuclear Medicine Service [5]: *The nuclear medicine department serves daily up to 50 patients and has a staff of about 15. Patients receive a small amount of a radioactive substance in the injection room before undergoing examinations such as bone and brain scans. Once injected, they emit radiation and are confined to the hot waiting room until the exams can be performed in one of the diagnostic rooms. The time patients must wait depends on the type of exam and how long the radioactive substance takes to propagate within the body and decay to the right level. After the exam, patients return to the hot waiting room until the level of radiation is below a certain threshold and they are harmless to others.*

Nurses typically accompany patients within the department, and the controls to prevent access by unauthorized people are not very rigorous. To address this problem, an application is developed so that patients are led automatically to where they must go, the correctness of their movements is verified, and access to restricted areas is controlled. The system uses RFID readers and tags to identify and locate patients. In addition, mobile devices will send appropriate messages to patients such as "move to the injection room" and "you are not allowed to enter this room." This will free nurses from having to guide patients as well as increase safety by reducing the number of people exposed to radiation.

In order to formalize this example we need to make the following assumptions:

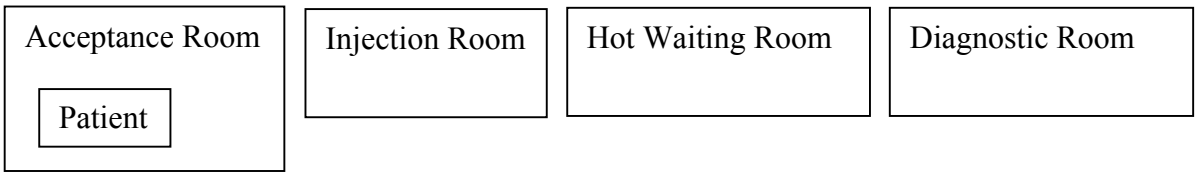
1. In this department, there are two kind of waiting rooms:

   Acceptance Room: waiting for injection;

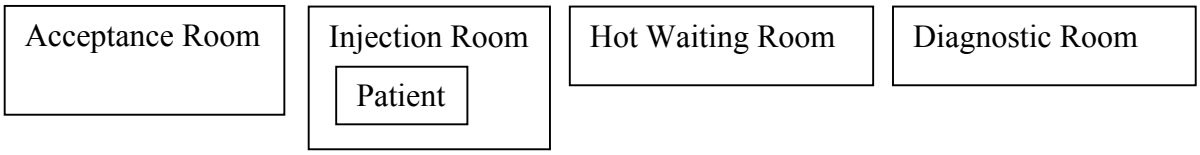
   Hot Waiting Room: waiting for diagnostic or the radiation threshold.

2. The Injection Room and Diagnostic Room can receive only one patient at a time.
3. Once admitted to this department, all the patients must finish all the steps before they can leave.

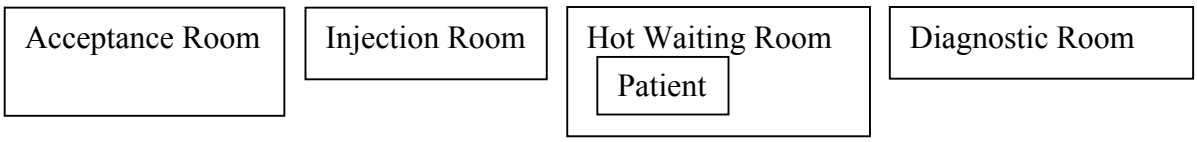
Step 1: enter the acceptance room



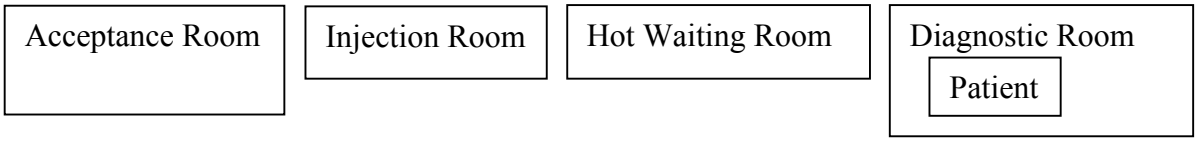
Step 2: enter the injection room, get injected



Step 3: enter the Hot waiting room, waiting for diagnostic



Step 4: enter the diagnostic room, get diagnostic



Step 5: re-enter the Hot waiting room, waiting for the radiation threshold, then leave

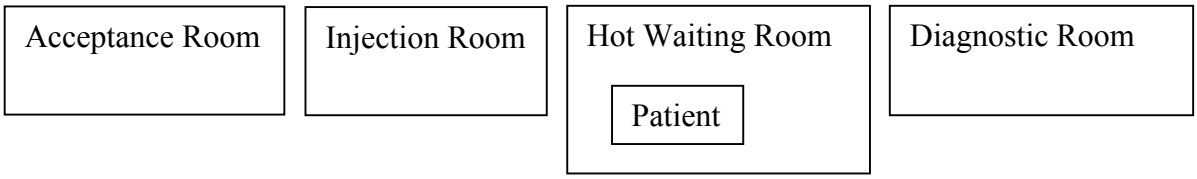


Figure 7: Nuclear medicine service: patient steps.

Figure 7 shows all the movement steps that one patient will have in this department. To simplify the presentation we use the following abbreviations:

Im: Injection Room;

Am: Acceptance/Waiting Room;

Wm: Hot Waiting Room;

Dm: Diagnostic Room;

Pati: Patient;

Injec: Get injection;

Diagn: Get diagnostic;

Callin: Called to Injection Room (by nurses or the application)

Calldi: Called to Diagnostic Room (by nurses or the application)

Callle: Called to leave Waiting Room (by nurses or the application)

4.3 Intuitive Design

We can build the Ambient language parser strictly according to the definition of Ambient Calculus [5]. If we use such a model to represent the steps from Figure 7, then we have the following process and reductions:

Patient ≜ (in Am. out Am. in Im. out Im. in Wm. out Wm. in Dm. out Dm. in Wm. out Wm. Pati) | Am[] | Wm[] | Im[] | Dm[]

≜ Am[(out Am. in Im. out Im. in Wm. out Wm. in Dm. out Dm. in Wm. out Wm. Pati)] | Wm[] | Im[] | Dm[] (Step1: Enter the acceptance room)

≜ Am[] | Wm[] | Im[(out Im. in Wm. out Wm. in Dm. out Dm. in Wm. out Wm. Pati)] | Dm[] (Step 2: Enter the injection room, get injected)

≜ Am[] | Wm[(out Wm. in Dm. out Dm. in Wm. out Wm. Pati)] | Im[] | Dm[]

(Step 3: Enter the Hot waiting room)

≙ Am[] | Wm[] | Im[] | Dm[(out Dm. in Wm. out Wm. Pati)]

(Step 4: Enter the diagnostic room)

≙ Am[] | Wm[(out Wm. Pati)] | Im[] | Dm[]

(Step 5: re-enter the Hot waiting room)

≙ Am[] | Wm[] | Im[] | Dm[] | Pati

One critical problem arises in the above process definition. Indeed, we just described the movement of patients, but all the other events during this process are not shown. We have no means to answer questions such as "when should the patient move in the Injection Room", "when should the patient leave the Hot waiting room"，"when will the radiation level of the patient reach the threshold", and so on.

The same problem exists in the Ambient Designer built by Coronato and De Pietro [4]. They developed their Ambient Designer to support a graphical modeling of Ambient processs, an example of which is shown Figure 7. Their Ambient Language parser is constructed strictly according to the definition of Ambient Calculus, same as the above intuitive design. Even if they set some flags in each locations, such as "Set.Busy" for the Injection Room, these flags are not able to describe the events during the patients' steps. Informally speaking, the event of the application advising the patient to move in the Injection Room cannot be shown.

To make up for this shortcoming, we therefore need to find a better textual model for our Ambient Calculus. We recall for this purpose the Communicating Sequential Processes (CSP) process algebra, which was designed for describing systems of

interacting components, and it is supported by an underlying theory for reasoning about them. In CSP the interacting components are independent self-contained entities with particular interfaces through which they interact with their environment. The interface of a process is described as a set of events. An event describes a particular kind of atomic, indivisible action that can be performed or suffered by the process. This kind of description is what we need for our Ambient Language parser. The details will be explained in next section.

4.4 Extended Design

In the previous section our straightforward ambient textual model and Coronato and De Pietro's prototype model are not perfect because they lack an "event expression mechanism". As such we inspire ourselves from classical process algebrae such as CSP [14, 15] and so we introduce the concept of “event” and “process” into our Ambient Language. In other words, we extend CSP to an Ambient Language. The steps of the patients shown in Figure 6 will thus be specified as follows:

Pati ≜ in.Am → Callin → out.Am → in.Im → Injec → out.Im → in.Wm → Calldi → out.Wm → in.Dm → Diagn → out.Dm → in.Wm → Callle → out.Wm → Pati

In this construction, we do not show the locations explicitly, but we offer instead an implicit notation that is based on agents entering or leaving those locations. That is, capabilities (see Figure 1) become the central mechanism for representing locations. In this way all the events during the evolution of the process could be expressed clearly, such as "when the event Callin happens, the patient move in the Injection Room", "when

the event Callle happens, the radiation level reaches the threshold and so the patient can leave", and so on.

Note in passing that several more processes (Doctor, InjectionRoom, Nurse, etc.) will need to be defined to complete the specification.

4.5 Design Analysis

Comparing the two designs presented above we conclude that the first is very formal and fully conforms to the syntax of the Ambient Calculus. The second design on the other hand does offer a restricted syntax, but is semantically equivalent and also easier to achieve in practice (since its implementation can be based on the already existing CSP module). For this reason we will use this (second) design.

Chapter 5 System Implementation

We are now ready to explain the design on the actual system.

5.1 Generating the Module Prototype

We use the PAT Module Generator to construct the Ambient Module. Figure 8 shows the GUI of the Module Generator.

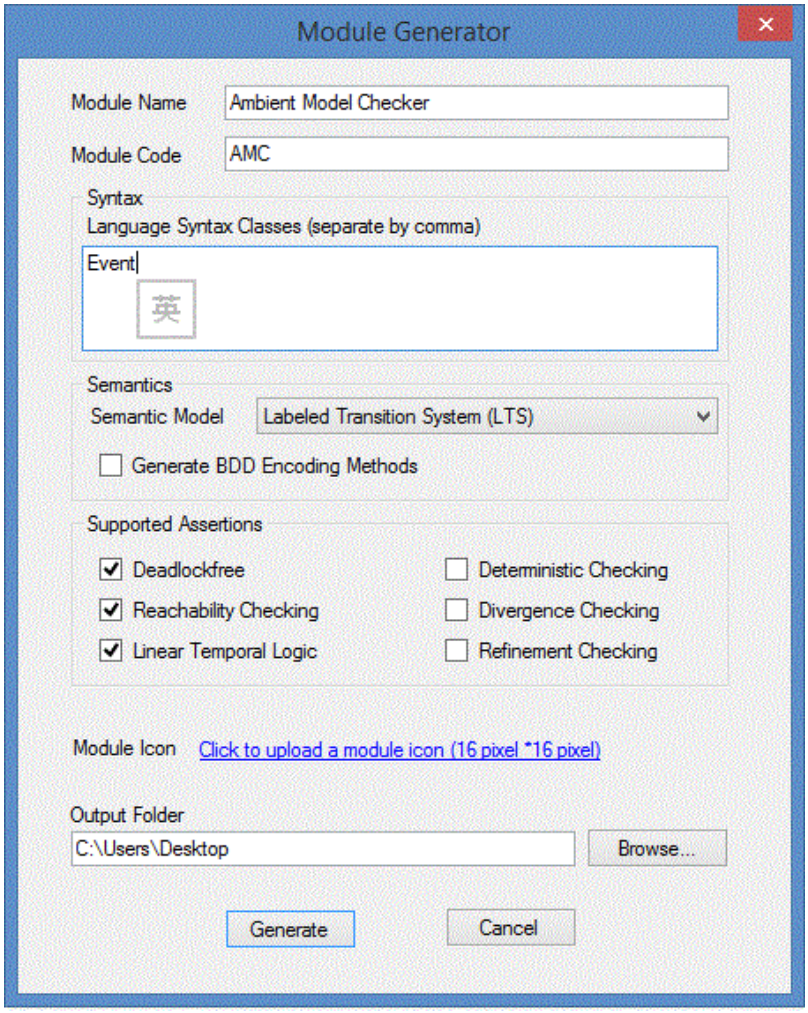


Figure 8: GUI of the PAT Module Generator.

First we name our module. The Module Code is an abbreviation of the module name, which will be used as extension for the input files of the generated module. In the Syntax panel, we can provide the language constructs of our new module. For example CSP supports interleave and parallel, so providing them will cause the generator to create the corresponding classes. In our case we just specify the basic "Event" class and we create all the other language constructs manually.

For the Semantics model, we chose the LTS (Labeled Transition System), since LTS offer a sufficiently comprehensive semantic model, and is also the semantic model for CSP (which we will extend). The generator will then automatically generate the corresponding classes.

For the Supported Assertions, we will choose the first three: "Deadlock free", "Reachability Checking", and "Linear Temporal Logic". The goal of this incipient level will be to only implement the first one or two assertions. Modifying the "Linear Temporal Logic" to conform to Ambient Module Checking is the subject of future research.

5.1.2 Working with Generated Code

The Generated code is a C# solution with two projects: PAT.Main and PAT.Module.AMC. The code is produced in the file "PAT3 Source.sln". "PAT.Main" is the GUI project, which we do not need to modify for our module development. This project includes a simple editor, which allows developers to input a model for simulation and verification. The project can be easily extended in the future depending on the needs of the developers.

"PAT.Module.AMC" is the module project, which includes "Assertions", "LTS", "Utility" and "ModuleFacade". As their name imply, "Assertions" contains all the assertions to be supported; "Utility" contains some utility functions; and "ModuleFacade" is the module interface class to communicate with the GUI classes. Finally "LTS" contains all the syntax classes, the specification class and the state interface class. Our work will therefore take place in "LTS".

5.2 Ambient Module Syntax

The syntax for the ambient module checker has two parts: one for the description of ambient calculus module, and one for the ambient logic specification.

5.2.1 Ambient Calculus Syntax

The process syntax shown in Figure 1 includes the following:

spatial constructs: **void, composition, replication, ambient, restriction.**

temporal constructs: **capability action, input action, output action**

To construct the textual syntax module we set the following literal rules to represent these constructs.

- All the **Process** names begin with a capital letter or underline and contains letters, numbers, and underline: ('A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*.
- All the **Location** names begin with a lower case letter or underline and contain letters, numbers, and underline: ('a'..'z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
- All the **Name or Variables** names begin with a letter or underline and contain letters, numbers, and underline: ('A'..'Z'|'a'..'z'|'_')('A'..'Z'|'a'..'z'|'0'..'9'|'_')*

- 'void' is the spatial construct **void 0**
- '*P* | *Q*' is the spatial construct **composition** P|Q
- '! *P*' is the spatial construct **replication** !P
- 'n[*P*]' is the spatial construct **ambient** M[P]

- '(vn) *P* ' is the spatial construct **restriction** (vn)P
- 'M.*P*' is the temporal construct **capability action** M.P
- '(n).*P*' is the temporal construct **input action** (n).P
- '<n>' is the temporal construct **output action** <M>

Capabilities are **name** (n), **enter into** (in M), **exit out of** (out M), **open** (open M), **null** (ε), and **path** (M.M). They will be included as specified, except for the following modifications:

1. Since the name defines an entity (such as location), there is no need to include it in **capabilities**.

2. When we talk about the capabilities, we normally mean something. So the **capability null** is useless and so we omit it.

In addition the capability **path** means one action following another, so we replace this construct with the ordered list of actions. Note as well that we only have three actions to consider: **enter**, **exit**, and **open**. Then we have:

- 'in Location' is the **capability** action **enter in** in M
- 'out Location' is the **capability** action **exit out of** out M
- 'open Location' is the **capability** action **open** M

5.2.2 Ambient Logic Syntax

The syntax of logical formulas is shown in Figure 4. At this incipient stage we only

specify a subset of these constructs, including the following:

- Propositional Logic: **True**, **negation**, **disjunction**
- Location Structure Logic: **void**, **composition**, **location**, **guarantee**, **placement**
- Temporal modality: **sometime modality** ($\Diamond \mathcal{A}$)
- Spatial modality: **somewhere modality** ($\blacklozenge \mathcal{A}$)

Then we set the following literal rules to represent these constructs.

- ' **T** ' is the propositional logic **True** (**T**)
- ' $\sim \mathcal{A}$ ' is the propositional logic **Negation** ($\neg\ \mathcal{A}$)
- ' $\mathcal{A} \vee \mathcal{B}$ ' is the propositional logic **Disjunction** ($\mathcal{A} \vee \mathcal{B}$)
- ' **0** ' is the structure logic **Void** (**0**)
- ' $\mathcal{A} \mid \mathcal{B}$ ' is the structure logic **Composition** ($\mathcal{A} \mid \mathcal{B}$)
- 'n[$\mathcal{A}$]' is the structure logic **Location** ($\eta\ [\mathcal{A}]$)
- ' $\mathcal{A} > B$ ' is the structure logic **Guarantee** ($\mathcal{A} \rhd \mathcal{B}$)
- ' $\mathcal{A}$ @ n' is the structure logic **Placement** ($\mathcal{A}$ @ η)
- '&T $\mathcal{A}$' is the temporal modality **sometime modality** ($\Diamond \mathcal{A}$)
- '&L $\mathcal{A}$ ' is the spatial modality **somewhere modality** ($\blacklozenge \mathcal{A}$)

5.3 Creating the Ambient Parser

As we said in Section 5.1.2, we need to create the parser classes manually to work with the generated code. We use ANTLR to create these classes. The resulting code is included in the files TreeLexer.cs, TreeParser.cs and TreeWalker.cs. The ANTLR code is

included in the files AMCTree.g (Section 5.3.1) and AmbientTreeWalker.g (Section 5.3.2).

5.3.1 Ambient Tree

The complete code is shown in Appendix A (this file must be saved with the same file name as the grammar name namely, "AMCTree.g" in our case). We first set the ANTLR options and declare all the tokens.

```
grammar AMCTree; // yields "AMCTreeLexer.cs" and "AMCTreeParser.cs"

options{
        output=AST;   // set the output as Abstract Syntax Tree, which is the "Tree.cs"
        ASTLabelType=CommonTree;
        memoize=true;
        language=CSharp2;   //Since we will work with the generated model in C# code
                            //we need to set Language as CSharp2. If we want to
                           //debug the syntax tree graphically, we need to set it as Java.
        }

tokens {
        SPEC_NODE;          //the root token

        LOCATION_NODE;          //these tokens for tree-like structure of locations
        SUBPARALLEL_NODE;
        SUBCASCADE_NODE;

        PROCESS_NODE;           //these tokens for the process definition
        PROC_DEF_NODE;
        SUBPROC_DEF_NODE;
        STATE_LOC_NODE;

        DEADLOCK_NODE;          //these two are for the assertion definition
        REACH_NODE;
        }
```

Next, we present the syntax of the ambient module. We start from the root token "SPEC_NODE" and continue as usual for such a definition.

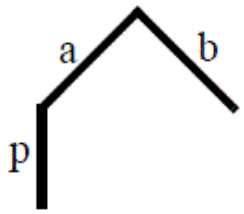
specDef:

((l+=location)+ (p+=process)+ (a+=assertion)+)
-> ^(SPEC_NODE $l+ $p+ $a+);

The parser will be divided into three parts: Ambient Structure, Process and Assertion.

(a) Ambient Structure Parser

To describe the Ambient Structure in text form we set up rules in two dimensions: parallel and cascade. For example, in the following edge-labeled tree, there are two contiguous locations, **a** and **b**, such that **b** has no sub-locations, and **a** has a sub-location **p**:



We convert this edge-labeled tree into the following text form:

#location cascade a[p];
#location parallel a|b;

We then use the following algorithm for specifying all the locations: Starting from the root,

Step 1: list one of the deepest cascade locations: ***#location cascade*** *a[p[m[...]]]*

Step 2: then in each location level, if there is any parallel locations not already mentioned, specify it: ***#location parallel*** *a|b|c|...;*

Step 3: in the previous step, if the parallel location has any sub-locations not already mentioned, repeat the first step until all the locations are specified.

The detail code is shown in Appendix A.

(b) Process Structure Parser

Same as for CSP, each process is defined by several states. In our case, we include the **movements**, **composition**, **events**, and **input/output actions**. The following is the process definition:

```
processDef:
        v=VAR '->' p=processDef
                                                    // rule for the "Event"
        | m = MOVE '.' l = LOC '->' p1 = processDef
                                                    // rule for the "Movements"
        | vo='<'VAROUT '>' '->' p2=processDef
                                                    // rule for the "output action"
        | vi='('VARIN')' '.' p3=processDef
                                                    // rule for the "input action"
        | s1 = subProcessDef '|' s2 = subProcessDef
                                                    // rule for the "composition"
        | s3 = subProcessDef
        | s=SKIP
        ;
```

In the above definition,

"VAR" represents the name of the "Event";

"MOVE" represents the action processes, including "in", "out" and "open";

"VAROUT" and "VARIN" represent variables.

The token "subProcessDef" offers two choices: the name of a sub-process or "void" as the end process. The detailed code is shown in Appendix A.

(c) Assertion Structure Parser

As mentioned in Section 5.1.1, our eventual goal is to implement the assertions "Deadlock free", "Reachability Checking" and "Linear Temporal Logic" (which will be actually renamed to "Ambient Logic"). So assertion structure contains the following:

assertion:

'#assert' v=VAR 'deadlockfree';'
// rule for "**deadlockfree**"

| '#assert' v1=VAR 'reaches' v2= stateLocationDef ';'
// rule for "**reachability**"

//| '#assert' v2=VAR '|-' v3 = formulaDef ';'
// rule for "**ambientLogic**"

;

When checking the specification "**deadlockfree**", we only need to specify the process under scrutiny, so in this assertion we put "VAR" as the name of the process. In the specification "**reachability**" we should check if a specific process can reach a specific state. Therefore, we also use "**stateLocationDef**", which could describe any process at any location (n[P|Q] for example). The detailed code is once more shown in Appendix A.

The semantics of the satisfaction operator for Ambient Logic is not included in this work, but is instead one goal of our future work on the matter.

(d) Parser Testing

The syntax presented by the parser is straightforward. We first tested our construction using the "**Nuclear Medicine Service**" example from Section 4.2. Figure 9 shows the

parse tree for the following input:

#location parallel am|im|dm|wm;
Pati = in.am -> Callin -> out.am -> in.im -> Injec -> out.im -> in.wm -> Calldi -> out.wm -> in.dm -> Diagn -> out.dm -> in.wm -> Callle -> out.wm -> Pati;
#assert Pati deadlockfree;

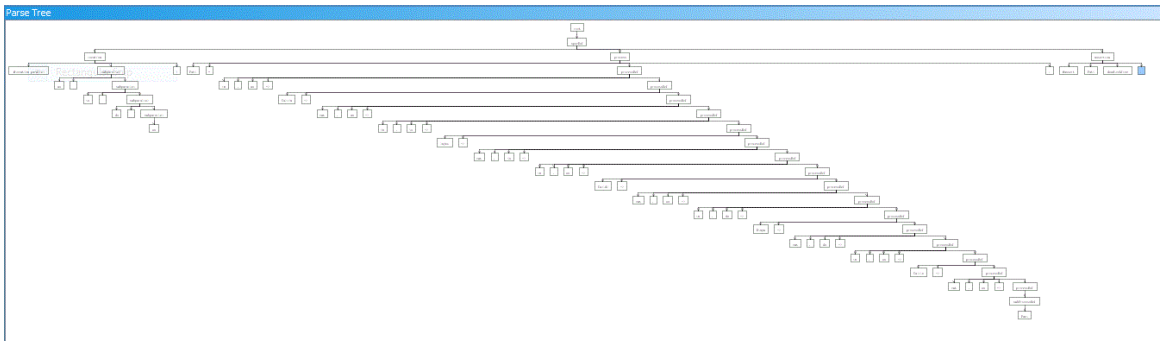


Figure 9: Nuclear medicine example: parse tree.

Since this is a very simple structure, we will also consider another example, as follows.

Senior Monitoring Service: *Some older people may have some threatening disease, like Myocardial infarction or Cerebral thrombosis. They need to be monitored with a monitoring bracelet all day. The monitoring bracelet will send the patient information to the monitoring center in real time. The information includes patient's location and health information. When the disease onsets, patient could be found and get help as soon as possible.*

Now, we assume there is one such patient who moves within the scope of hospital, home and drugstore. This hospital has one *nuclear medicine department* (as mentioned in the previous chapter). Then we could describe the ambient structure as in Figure 10, where we use the following abbreviations: "hos" for hospital, "nuc_dept" for nuclear medicine

department, "hom" for home and "drug" for drugstore.

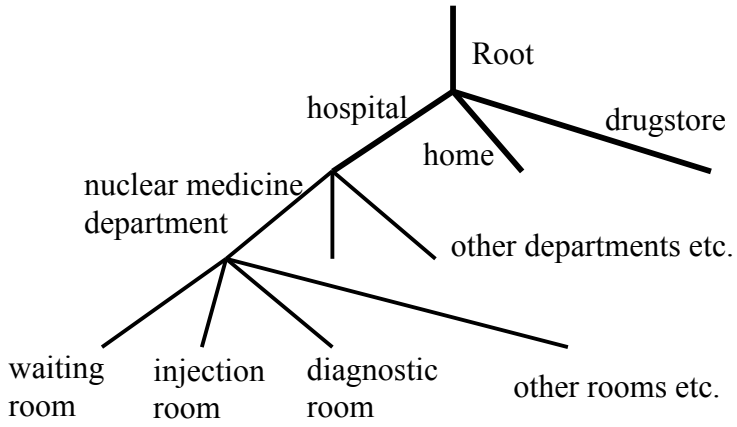


Figure 10: Senior monitoring service: ambient structural map.

The above ambient structure could be described in the following text form as the parsing rules shown in section **(a)**.

 # location cascade hos[nuc_dept[wm]];
#location parallel am|im|dm|wm;
#location parallel hom|drug|hos;

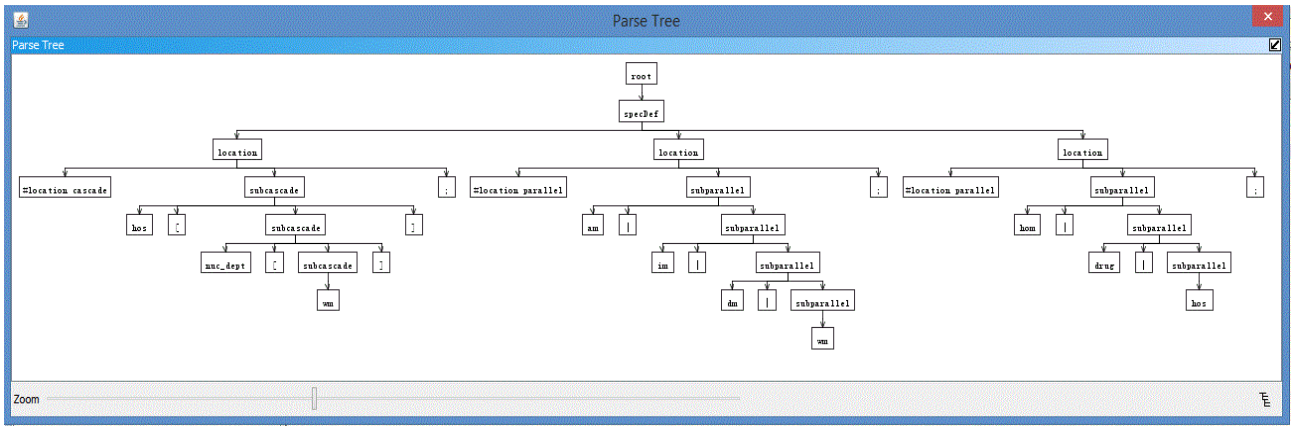


Figure 11: Senior monitoring service: ambient structure parse tree.

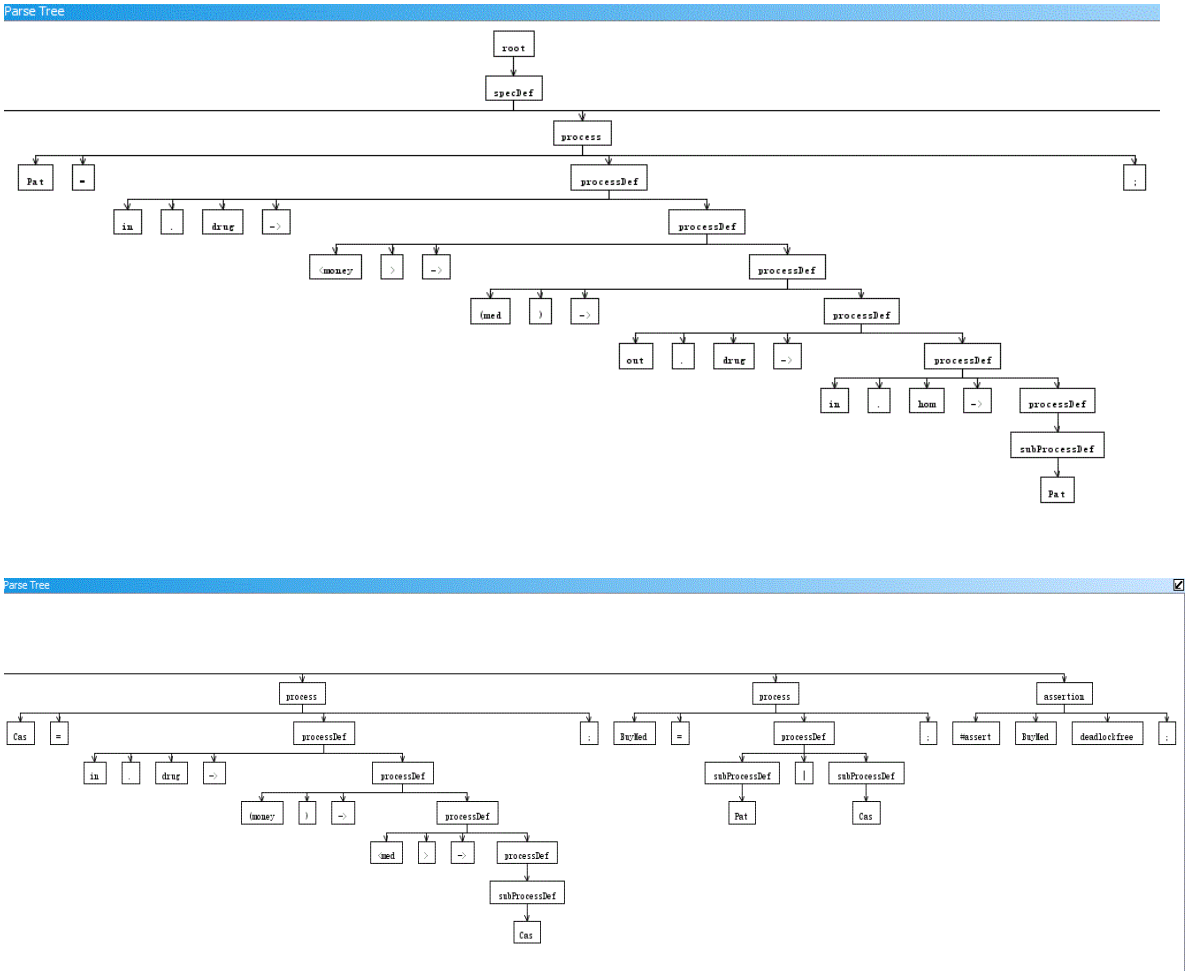


Figure 12: Senior monitoring service: process and assertion parse trees.

We will now consider the process of the patient buying medicine after he leaves the nuclear medicine department from the hospital. At first, the patient should enter the pharmacy, then pay for the medicine they need. At the same time, the cashier in this pharmacy will receive the money and hand over the medicine. These two processes are arranged in a composition relation. We thus have the following textual description for the whole process ("Cas" is the abbreviation of Cashier, "med" is the abbreviation of medicine):

Pat = in.drug -> <money> -> (med).Pat -> out.drug -> in.hom -> Pat;
Cas = in.drug -> (money).Cas -> <med> -> Cas;
BuyMed = Pat|Cas;
#assert BuyMed deadlockfree;

The resulting parse tree is shown in Figure 12.

5.3.2 Ambient Tree Walker

Now that the parser grammar for the Ambient module is in place we can introduce the associated tree grammars. This grammar is defined in the separate file "AMCTreeWalker.g".

Similar to the parser grammars, the tree grammar begin with the grammar header and some options:

```
tree grammar AMCTreeWalker;      // yields AMCTreeWalker.cs
options {
    tokenVocab=AMCTree;          // read token types from AMCTree.tokens file
    ASTLabelType = CommonTree;
    language=CSharp2;
    memoize=true;
    }
```

The *tokenVocab* option indicates that the tree grammar should preload the token names and associated token types defined in AMCTree.tokens, which is generated after processing AMCTree.g. When we program tree grammar, we want it to use the same token IDs and token types as used by the parser grammar [13].

We then define rules for all of the tree grammars. Each expression rule in the parser grammar corresponds to a single rule in the tree grammar. Most of their semantics are obvious. Starting from the root token "specDef" we proceed with three parts: Location, Process and Assertion.

(a) Tree Grammar: Location

Since we describe the ambient structure in two dimensions, there might be some overlap between them. We should get rid of the overlap and save location information into memory.

First we define a class to hold the information for locations. In our case, we defined a class name "Location", which includes one pointer to parent location, one list to hold sub-locations, and another list to hold the processes located here. We also define a hash table to keep the location information. In our case, we declared a String Dictionary named "LocationDatabase".

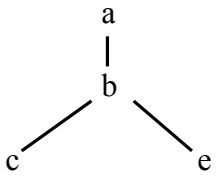
Then we start the three steps mentioned in Section 5.3.1**.**

Step 1: list one of the deepest cascade locations: ***#location cascade*** *a[p[m[...]]];*

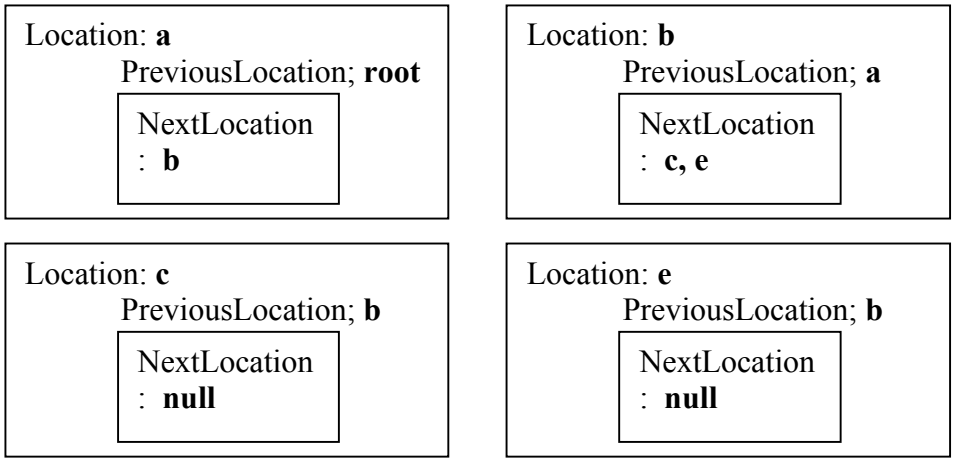
When the parser reads the first location name, we create a new location in the "LocationDatabase" and set it as the "LastLocation". Then once the parser reads one location name, we will search it in the "LocationDatabase". If not found, then a new location is created and the previous location pointer "LastLocation" is set to it. If on the other hand the location name already exists, then we will get this location and add the "LastLocation" into its "NextLocation" list. We then set "LastLocation" to this location.

Consider for example the following ambient structure input:

#location cascade *a[b[c]];*
#location cascade *b[e];*



After Step 1 the following location objects will be in place:



Step 2: then in each location level, if there is any parallel locations not already mentioned, specify it: ***#location parallel*** *a|b|c|...;*

The parallel location rule, is only a supplement. Same as in the previous step, we add the location into "NextLocation" list and set the "PreviousLocation" accordingly. However, there must be one location already entered in the listed parallel locations, and it must be listed last. Indeed, note that our parser read the textual input from back to front, and so we need one existing location to know which is the "PreviousLocation" of these parallel locations.

Step 3, in the previous step, if the parallel location has any sub-locations not already mentioned, repeat the first step until all the locations are considered.

We will eventually get all the locations entered into the "LocationDatabase". When needed, the location map can also be printed.

(b) Tree Grammar: Process and Assertion

Each **Process** and **Assertion** will have a corresponding special module component (specifically, a class). We will present these classes in next section. The whole "AMCTreeWalker.g" file is shown in Appendix B.

(c) Tree Grammar: Parser Test

We tested our construction using the "**Senior Monitoring Service**" example from Section 5.3.1. First, we need to input the following text description to the model checker, which is shown in Figure 13. Then press the key "**Check Grammar**", we get the grammar walker test result, which is shown in Figure 14.

#location cascade hos[nuc_dept[wm]];
#location parallel am|im|dm|wm;
#location parallel hom|drug|hos;
Pat = in.drug -> <money> -> (med).Pat -> out.drug -> in.hom -> Pat;
Cas = in.drug -> (money).Cas -> <med> -> Cas;
BuyMed = Pat|Cas;
#assert BuyMed deadlockfree;

Input text description:

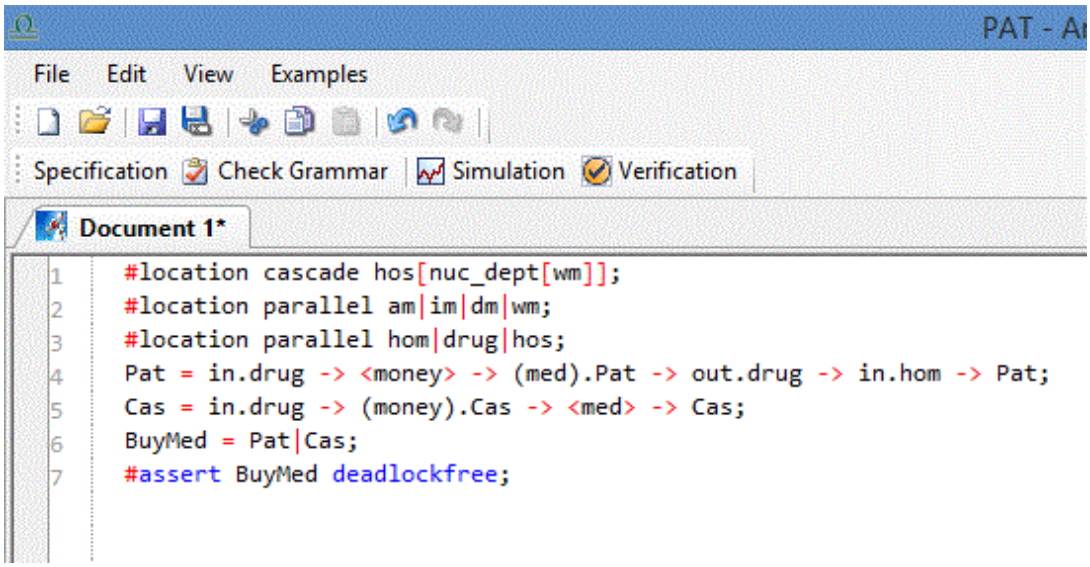


Figure 13: Grammar walker testing input window.

Output parsing tree:

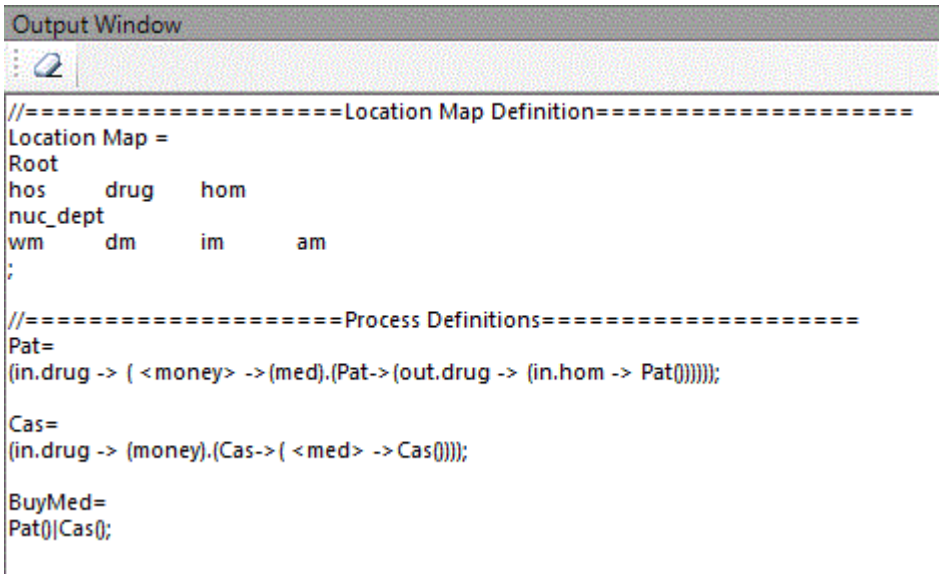


Figure 14: Grammar walker testing result.

There are several sections in the results window: The Location Map definition shows the location map. Every row means one level, and the location "**Root**" always lies

at the top level. In the Process Definitions Section, every process is shown in the "state -> process" form.

5.4 Module Components

So far we have created the three parser files for Ambient Module Checker. Next, we need to complete all the language construct representations, including Definition, DefinitionRef, EventPrefix, Composition, Process, Location, MoveInLocation, MoveOutLocation, OpenLocation, Variables, InputVariable, OutputVariable, Void, Skip and Stop. There are two classes created for this purpose.

5.4.1 Configuration.cs

This class is generated by the Module Generator. It represents the semantic model of the language. When we run the model checker, each step will yield one configuration object. The method "MakeOneMove" in this class will then be invoked by the other components to determine the proper action.

5.4.2 Specification.cs

This class is also generated by the Module Generator. It is the controlling class, which communicates with ModuleFascade.cs and LTS. To complete the code, we need to complete the method "ParseSpec()". To invoke the parser and analyze the input model we added the following code in this method.

```
ICharStream input = new ANTLRStringStream(spec);
```

```
                    // create a CharStream that reads from standard input
AMCTreeLexer lex = new AMCTreeLexer(input);
                    // create a lexer that feeds off of input CharStream
CommonTokenStream tokens = new CommonTokenStream(lex);
                    // create a buffer of tokens pulled from the lexer
AMCTreeParser parser = new AMCTreeParser(tokens);
                    // create a parser that feeds off the tokens buffer
 AMCTreeParser.specDef_return r = parser.specDef();
```

Additionally, the location database "LocationDatabase" and transit location "LastLocation", which are mentioned in section 5.3.2, should be declared in this file.

5.4.3 Process Definition Classes

We believe that the language structure representations could be divided into several categories. On one hand basic structure representations include "EventPrefix.cs", "Composition.cs", "Void.cs", and on the other hand location movement representations include "Location.cs", "MoveInLocation.cs", "MoveOutLocation.cs", "OpenLocation.cs". We will illustrate in what follows these two groups of representations.

(a) Base Class "Process"

Before talking about these structure classes we have to introduce the “Process” base class, defined in the file "Process.cs". All the other structure classes are derived from this base class.

```
public abstract class Process
{       public string ProcessID;
            // Set name to every step the process moved
```

```
        public Location LocationInclude;
                // Keep the "Location" where the current process locate
        public abstract void MoveOneStep(Valuation GlobalEnv, List<Configuration> list);
                // This method returns all the possible moves of the current process
                // <param name="GlobalEnv">The current global valuation</param>
                // <param name="list">The list of steps to be returned.</param>
        ... ...
}
```

The whole class file is shown in Appendix C.

(b) Basic Structure Representations

"Void.cs" has the simplest structure. When we construct a void process, we just need to do 2 things: set the **processID** as "void" to let our checker know this is a void process, and do nothing in the method "**MoveOneStep**" (that is, we skip one step).

"EventPrefix.cs" represent the occurrence of a simple event such as the patient being called by the nurse. We need to record the name of this event as a character string, and keep the process going. Therefore in the method "**MoveOneStep**", we create a new configuration for each "EventPrefix" step.

```
public class EventPrefix : Process
{       public string Event;            // set the name of "Event"
        public Process Process;         // keep the following process

        public EventPrefix(string previous, Process process)
                                        // constructor to initialize the object
        {       Event = previous;
                Process = process;
                ProcessID = ... ...;        }
```

```
        public override void MoveOneStep(Valuation GlobalEnv, List<Configuration> list)
                                        // returns all the possible moves of the current process
        {       System.Diagnostics.Debug.Assert(list.Count == 0);
                list.Add(new Configuration(Process, Event, GlobalEnv, false));
                                        // create a new configuration for this step   }
        ... ...
}
```

The whole class file is shown in Appendix D.

"Compostion.cs" is very similar to the interleave operator in CSP. In the method "MoveOneStep", we just need to alternate between the two processes, so that we can traverse all the possible states. The class file is shown in Appendix E.

(c) Location Movement Representations

We created the class **Location** (file "Location.cs") to build the topology of ambients.

```
public class Location
{       public string LocationID;                   // set the LocationID to name this location
        public Location PreviousLocation;           // set the pointer to parent Location
        public List<Location> NextLocation;         // keep the child locations in this list
        public List<Process> ProcessIncludeList; // keep the processes which is locating here

        public Location(string locaname) {... ...}
        public Location(string locname, Location NexLoc) {... ...}
                                        // constructor to initialize the object
}
```

"MoveInLocation.cs" describes the action that one process '**P**' moves in one Location '**n**'. We set the Location **n** to the variable "LocationInclude" in process **P**. Then

add this process **P** into the Location **n**'s process list "ProcessIncludeList".

"MoveOutLocation.cs" describes the action that one process '**P**' moves out of one Location '**n**'. We remove the Location **n** from the **P**'s variable "LocationInclude". Then we remove process **P** from the Location **n**'s process list "ProcessIncludeList".

"OpenLocation.cs" describes the action that deletes one Location from our module. We need to remove this Location from our "LocationDatabase" and at the same time clear the **P**'s variable "LocationInclude".

The class files are shown in Appendices F to I.

Chapter 6 Conclusion

An ambient model checker will eventually support reasoning about the safety properties of ambient systems. That is, given a system model written in the Ambient Calculus (but not using replication) and the properties specified as Ambient Logic formulas, our ambient model checker could determine whether this system model satisfies those properties.

This paper does not present a complete ambient checker, but we have laid a solid basis for this pursuit. Indeed, we constructed the syntax part of an ambient model checker, which allows the formal specification of ambient systems using the Ambient Calculus. We then constructed classes for Ambient processes and their actions. Finally,

we have provided the definition for the Ambient Logic syntax, meaning that this project can only parse ambient assertions but cannot yet verify these assertions against the given process. In all we can now parse processes and locations specified using the Ambient Calculus, and we can specify properties in Ambient Logic. The implementation of the actual model checking algorithm is the subject of our future work on the matter.

We constructed this system as a PAT module and so we will continue to operate within the PAT framework while developing this algorithm. This will require a deeper investigation of the way PAT implements semantic analysis. We believe that we will be able to adapt this module toward model checking Ambient Calculus [7]. We further believe that an ambient model checker will be a useful tool, especially in the area of pervasive computing.

References

[1] **ANTLR Web site**. http://www.antlr.org

[2] L.Cardelli, A.D. Gordon. **Anytime, Anywhere: Modal Logics for Mobile Ambients**. The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 365-377, Microsoft Research, New York, NY, USA, 2000.

[3] L. Cardelli, A.D. Gordon: **Ambient Logic.** Mathematical Structures in Computer Science (2006).

[4] Antonio Coronato and Giuseppe De Pietro. **Formal Design of Ambient Intelligence Applications**. Computer 43(12): 60-68 (2010).

[5] L. Cardelli, A.D. Gordon: **Mobile Ambients**. Theoretical Computer Science 240:177-213 (2000).

[6] E. M. Clarke, O. Grumberg, D. A. Peled. **Model Checking**. MIT Press, 1999.

[7] Witold Charatonik, Silvano Dal Zilio, A.D. Gordon. **Model Checking Mobile Ambients**. Theoretical Computer Science 308:277-331 (2003).

[8] Jin Song Dong, Jun Sun and Yang Liu. **Build Your Own Model Checker in One Month**. The 35th International Conference on Software Engineering (ICSE 2013), Tutorial, San Francisco, CA, USA, May 18 - 26, 2013 .

[9] E. Allen Emerson. **Temporal and Modal Logic**. Handbook of the Theoretical Computer Science, vol. B, pages 995-1072, 1990.

[10] Yang Liu, Jun Sun and Jin Song Dong. **PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers**. The 22nd Annual International

Symposium on Software Reliability Engineering (ISSRE 2011), pages 190-199, Hiroshima, Japan, Nov 29 - Dec 2, 2011.

[11] **PAT Web site**. http://www.comp.nus.edu.sg/~pat/

[12] T. Parr. **Getting started with ANTLR 4**. https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Getting+Started+with+ANTLR+v4

[13] Terence Parr. **The Definitive ANTLR Reference: Building Domain-Specific Languages**, Pragmatic Bookshelf, 2007.

[14] A.W. Roscoe. **The Theory and Practice of Concurrency**, Prentice Hall, 1997.

[15] Steve Schneider. **Concurrent and Real-time Systems: The CSP Approach**, John Wiley & Sons, 2000.

[16] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, Étienne André. **Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP**. The ACM Transactions on Software Engineering and Methodology (TOSEM), 22(1):3:1-3:29 (2013).

Appendices

Appendix A: AmbientTree.g

```
grammar AMCTree;

options
{
	output=AST;
	ASTLabelType=CommonTree;
	memoize=true;
	language=
	//Java;
	CSharp2;
	}

tokens {
	SPEC_NODE;
	LOCATION_NODE;
	SUBPARALLEL_NODE;
	SUBCASCADE_NODE;
	PROCESS_NODE;
	PROC_DEF_NODE;
	SUBPROC_DEF_NODE;
	STATE_LOC_NODE;
	DEADLOCK_NODE;
	REACH_NODE;
}

@namespace   {PAT.AMC.LTS}
@header{
	//using System;
	using PAT.Common;
	}

@members
{
  public Stack<String> paraphrases = new Stack<String>();
  public override String GetErrorMessage(RecognitionException e, String[] tokenNames)
   {
	string msg = null;
	if ( e is NoViableAltException ) {
```

```
                msg = "Invalid Symbol: '" + e.Token.Text + "'";
                }
        else
        {
                msg = base.GetErrorMessage(e, tokenNames);
                }
      if (paraphrases.Count > 0)
      {
        String paraphrase = (String)paraphrases.Peek();
        msg = msg + " " + paraphrase;
        }
      msg = msg + "\n" + "Line " + e.Token.Line + ", Column " +
e.Token.CharPositionInLine;
        return msg;
   }
}
@rulecatch {
catch (RecognitionException re)
{
    String sError=GetErrorMessage(re, tokenNames);
    throw new ParsingException(sError, re.Token);
        }
}

public
specDef
@init  { paraphrases.Push(" AMC input analysis");  }
@after { paraphrases.Pop(); }
        :
        ((l+=location)+ (p+=process)+ (a+=assertion)+)
        -> ^(SPEC_NODE $l+ $p+ $a+)
        ;

location
@init  { paraphrases.Push(" Location analysis");  }
@after { paraphrases.Pop(); }
        :
        '#location parallel' sp = subparallel ';'
        -> ^(LOCATION_NODE $sp)
        | '#location cascade' sc = subcascade ';'
        -> ^(LOCATION_NODE $sc)
        ;

subcascade
@init  { paraphrases.Push(" Cascade location analysis");  }
@after { paraphrases.Pop(); }
        :
```

```
        l1 = LOC '[' s1 = subcascade ']'
        -> ^(SUBCASCADE_NODE $l1 $s1)
        | l2 = LOC
        -> ^(SUBCASCADE_NODE $l2)
        ;

subparallel
@init  { paraphrases.Push(" Parallel location analysis");  }
@after { paraphrases.Pop(); }
        :
        l1 = LOC '|' s1 = subparallel
        -> ^(SUBPARALLEL_NODE $l1 $s1)
        | l2 = LOC
        -> ^(SUBPARALLEL_NODE $l2)
        ;

process
@init  { paraphrases.Push(" process equation analysis");  }
@after { paraphrases.Pop(); }
        :
        v=VAR '=' p=processDef ';'
        -> ^(PROCESS_NODE $v $p)
        ;

processDef
@init  { paraphrases.Push(" process definition analysis");  }
@after { paraphrases.Pop(); }
        :
         v=VAR '->' p=processDef
          -> ^(PROC_DEF_NODE $v $p)
        | m = MOVE '.' l = LOC '->' p1 = processDef
          -> ^(PROC_DEF_NODE $m $l $p1)
        | vo=VAROUT '>' '->' p2=processDef
          -> ^(PROC_DEF_NODE $vo $p2)
        | vi=VARIN')' '.' p3=processDef
          -> ^(PROC_DEF_NODE $vi $p3)
        | s1 = subProcessDef '|' s2 = subProcessDef
          -> ^(PROC_DEF_NODE $s1 $s2)
        | s3 = subProcessDef
          -> ^(PROC_DEF_NODE $s3)
        | s=SKIP
          -> ^(PROC_DEF_NODE $s)
        ;

subProcessDef
@init  { paraphrases.Push(" process subDefinition analysis");  }
@after { paraphrases.Pop(); }
```

```
        :
        v1= VAR
        -> ^(SUBPROC_DEF_NODE $v1)
        |'!' v2= VAR
        -> ^(SUBPROC_DEF_NODE $v2)
        | v3= VOID
        -> ^(SUBPROC_DEF_NODE $v3)
        ;
stateLocationDef
@init  { paraphrases.Push(" State with Location analysis");  }
@after { paraphrases.Pop(); }
        :
        l=LOC '[' s = stateLocationDef ']'
        -> ^(STATE_LOC_NODE $l $s)
        | s1 = subProcessDef '|' s2= subProcessDef
        -> ^(STATE_LOC_NODE $s1 $s2)
        | s3 = subProcessDef
        -> ^(STATE_LOC_NODE $s3)
        ;

assertion
        :
          '#assert' v=VAR 'deadlockfree';'
        -> ^(DEADLOCK_NODE $v)
        | '#assert' v1=VAR 'reaches' v2= stateLocationDef ';'
        -> ^(REACH_NODE $v1 $v2)
        //| '#assert' v2=VAR '|-'
        ;

MOVE: ('in'|'out'|'open')
        ;
VARIN: ('(')('a'..'z'|'A'..'Z'|'0'..'9'|'_')+
        ;
VAROUT: ('<')('a'..'z'|'A'..'Z'|'0'..'9'|'_')+
        ;
VOID  : 'Void'
        ;
SKIP   : 'Skip'
        ;

VAR   : ('A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
        ;

LOC   : ('a'..'z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
        ;

COMMENT
```

```
        :  '/*' ( options {greedy=false;} : . )* '*/' {$channel=Hidden;}
        ;

LINE_COMMENT
        :  '//' ~('\n'|'\r')* '\r'? '\n' {$channel=Hidden;}
        ;

WS      :  (' ' | '\t' | '\n' | '\r' | '\f')+ {$channel = Hidden;};
```

Appendix B: AmbientTreeWalker.g

```
tree grammar AMCTreeWalker;

options
{
        tokenVocab=AMCTree;
        ASTLabelType = CommonTree;
        language=CSharp2;
        memoize=true;
}

@namespace
{PAT.AMC.LTS}

@header{
//using System.Collections.Generic;
using System.IO;
using PAT.Common;
using PAT.Common.Classes.Assertion;
using PAT.Common.Classes.ModuleInterface;
using PAT.AMC.Assertions;
}

@members
{
        public static Stack<String> paraphrases = new Stack<String>();
        private List<DefinitionRef> defList = new List<DefinitionRef>();
        private string options="";
        public Specification Spec;

        public override String GetErrorMessage(RecognitionException e, String[]
tokenNames)
        {
                string msg = null;
```

```
                if ( e is NoViableAltException ) {
                        msg = "Invalid Symbol: '" + e.Token.Text + "'";
                }
                else
                {
                        msg = base.GetErrorMessage(e, tokenNames);
                }

        if (paraphrases.Count > 0)
        {
                        string paraphrase = (string)paraphrases.Peek();
                        msg = msg + " " + paraphrase;
        }

        msg = msg + "\n" + "Line " + e.Token.Line + ", Column " +
e.Token.CharPositionInLine;

        return msg;
    }

    public void program(string option){
        options = option;
        specDef();
        foreach(var dref in defList){
                dref.Def = Spec.DefinitionDatabase[dref.Name];
        }
    }
}

@rulecatch {
catch(RecognitionException re){
        string sError=GetErrorMessage(re, tokenNames);
        throw new ParsingException(sError, re.Token);
}
}

specDef
@init  { paraphrases.Push(" AMC input analysis");  }
@after { paraphrases.Pop(); }
:

         ^(SPEC_NODE       (l=location{})+
                        (p=process{Spec.DefinitionDatabase.Add(p.Name, p);})+
                        (a=assertion)+
         )
;
```

```
location
@init  { paraphrases.Push("location map cascade analysis");  }
@after { paraphrases.Pop(); }
        :
        ^(LOCATION_NODE sp = subparallel{})
        |^(LOCATION_NODE sc = subcascade{})
        ;

subparallel
@init  { paraphrases.Push("location map parallel analysis");
        Location loc;  }
@after { paraphrases.Pop(); }
        :
        ^(SUBPARALLEL_NODE l1 = LOC s1 = subparallel{
                if(!Spec.LocationDatabase.ContainsKey($l1.Text))
                {
                        loc = new Location($l1.Text);
                        loc.PreviousLocation = Spec.LastLocation.PreviousLocation;
                        Spec.LastLocation.PreviousLocation.NextLocation.Add(loc);
                        Spec.LocationDatabase.Add($l1.Text,loc);
                }
                else
                {
                        Spec.LastLocation =
Spec.LocationDatabase[$l1.Text].PreviousLocation;
                }})
        |^(SUBPARALLEL_NODE l2 = LOC{
                if(Spec.LocationDatabase.ContainsKey($l2.Text))
                {
                        Spec.LastLocation = Spec.LocationDatabase[$l2.Text];
                }})
        ;

subcascade
@init  { paraphrases.Push("location map cascade analysis");
        Location loc;  }
@after { paraphrases.Pop(); }
        :
        ^(SUBCASCADE_NODE l1 = LOC s1 = subcascade{
                if(!Spec.LocationDatabase.ContainsKey($l1.Text))
                {
                        loc = new Location($l1.Text, Spec.LastLocation);
                        Spec.LastLocation.PreviousLocation = loc;
                        Spec.LastLocation = loc;
                        Spec.LocationDatabase.Add($l1.Text,loc);
                }
                else
```

```
                {
                        loc = Spec.LocationDatabase[$l1.Text];
                        loc.NextLocation.Add(Spec.LastLocation);
                        Spec.LastLocation.PreviousLocation = loc;
                        Spec.LastLocation = loc;
                }})
        |^(SUBCASCADE_NODE l2 = LOC{
                if(!Spec.LocationDatabase.ContainsKey($l2.Text))
                {
                        loc = new Location($l2.Text);
                        Spec.LastLocation = loc;
                        Spec.LocationDatabase.Add($l2.Text,loc);
                }})
        ;
process returns[Definition pDef]
@init  { paraphrases.Push(" process equation analysis");  }
@after { paraphrases.Pop(); }
        :
        ^(PROCESS_NODE v=VAR p=processDef{pDef= new Definition($v.Text,null,
p);})
        ;

processDef returns[Process proc]
@init  { paraphrases.Push(" process definition analysis");
         string str;
         Location loc; }
@after { paraphrases.Pop(); }
        :
          ^(PROC_DEF_NODE v=VAR p=processDef{proc = new EventPrefix($v.Text,
p);})
        | ^(PROC_DEF_NODE m = MOVE l = LOC p1 = processDef {
                loc = Spec.LocationDatabase[$l.Text];
                switch ($m.Text)
                        {
                                case "in":
                                        proc = new MoveInLocation(loc, p1);
                                        break;
                                case "out":
                                        proc = new MoveOutLocation(loc, p1);
                                        break;
                                case "open":
                                        proc = new OpenLocation(loc, p1);
                                        break;
                                default:
                                        break;
                        }
                })
```

```
        | ^(PROC_DEF_NODE vo = VAROUT p2 = processDef {
                str = $vo.Text;
                string sub = str.Substring(1, str.Length-1);
                Variable outputVar = new Variable(sub);
                proc = new OutputVariable(outputVar, p2);})
        | ^(PROC_DEF_NODE vi = VARIN p3 = processDef {
                str = $vi.Text;
                string sub = str.Substring(1, str.Length-1);
                Variable inputVar = new Variable(sub);
                proc = new InputVariable(inputVar, p3);})
        | ^(PROC_DEF_NODE s1=subProcessDef s2 = subProcessDef{
                proc = new Composition(s1, s2);
                })
        | ^(PROC_DEF_NODE s3=subProcessDef{
                proc = s3;
                })
        | ^(PROC_DEF_NODE s=SKIP{proc = new Skip();})
        ;

subProcessDef returns[DefinitionRef subproc]
@init  { paraphrases.Push(" process subDefinition analysis");  }
@after { paraphrases.Pop(); }
        :
          ^(SUBPROC_DEF_NODE v1= VAR{
                DefinitionRef prec = new DefinitionRef($v1.Text);
                defList.Add(prec);
                subproc = prec;
                })

        | ^(SUBPROC_DEF_NODE v3=VOID{
                DefinitionRef VoidRef = new Void();
                subproc = VoidRef;})
        ;
stateLocationDef returns[Process stateproc]
@init  { paraphrases.Push(" State with Location analysis");  }
@after { paraphrases.Pop(); }
        :
          ^(STATE_LOC_NODE l=LOC s= stateLocationDef {
                Location loc = Spec.LocationDatabase[$l.Text];
                stateproc = new MoveInLocation(loc, s);
                })
        | ^(STATE_LOC_NODE s1=subProcessDef s2 = subProcessDef{
                DefinitionRef p1 = s1;
                defList.Add(p1);
                DefinitionRef p2 = s2;
                defList.Add(p2);
                stateproc = new Composition(p1, p2);
```

```
                })
        | ^(STATE_LOC_NODE s3=subProcessDef{
                stateproc = s3;
                })
        ;
assertion
@init{AssertionBase ass;
string assString;
string statestring;}
        :
          ^(DEADLOCK_NODE  v=VAR{
          var dref1 = new DefinitionRef($v.Text);
          defList.Add(dref1);
          ass = new AMCAssertionDeadLock(dref1); ass.AssertToken = $v.Token;
        assString = ass.ToString();
        if(Spec.AssertionDatabase.ContainsKey(assString))
        {
                throw new ParsingException("Assertion " + assString + " is defined
already!", ass.AssertToken);
        }else{
                Spec.AssertionDatabase.Add(assString, ass);
        }
          })
        | ^(REACH_NODE v1=VAR s = stateLocationDef{
        var dref2 = new DefinitionRef($v1.Text);
        defList.Add(dref2);
        statestring = s.ProcessID;
        ass = new AMCAssertionReachability(dref2, statestring); ass.AssertToken =
$v1.Token;
        assString = ass.ToString();
        if(Spec.AssertionDatabase.ContainsKey(assString))
        {
                throw new ParsingException("Assertion " + assString + " is defined
already!", ass.AssertToken);
        }else{
                Spec.AssertionDatabase.Add(assString, ass);
        }
        })
        ;
```

Appendix C: Process.cs

```
using System.Collections.Generic;
using PAT.Common.Classes.Expressions;
```

```
using PAT.Common.Classes.Expressions.ExpressionClass;

namespace PAT.AMC.LTS{
        public abstract class Process
        {       public string ProcessID;
                public Location LocationInclude;
                                // returns all the possible moves of the current process
                public abstract void MoveOneStep(Valuation GlobalEnv,
List<Configuration> list);
        }
}
```

Appendix D: Void.cs

```
using System;
using System.Collections.Generic;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.Common.Classes.Ultility;

namespace PAT.AMC.LTS
{
   public sealed class Void : DefinitionRef
   {
      public Void():base("void")
      {
          ProcessID = "void";
      }

      public override void MoveOneStep(Valuation GlobalEnv, List<Configuration> list)
      {
         System.Diagnostics.Debug.Assert(list.Count == 0);
      }

      public override string ToString()
      {
         return "void";
      }
   }
}
```

Appendix E: EventPrefix.cs

```
using System.Collections.Generic;
using System.Text;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.Common.Classes.LTS;
using PAT.Common.Classes.ModuleInterface;
using PAT.Common.Classes.Ultility;
using PAT.AMC.Assertions;

namespace PAT.AMC.LTS{
public class EventPrefix : Process
{
        public string Event;
    public Process Process;

     //TODO: constructor to initialize the object
        public EventPrefix(string previous, Process process)
    {
        Event = previous;
        Process = process;
        ProcessID = DataStore.DataManager.InitializeProcessID( Event.ToString() +
Constants.EVENTPREFIX + Process.ProcessID);
    }

        public override void MoveOneStep(Valuation GlobalEnv, List<Configuration>
list)
    {
        System.Diagnostics.Debug.Assert(list.Count == 0);
        list.Add(new Configuration(Process, Event, GlobalEnv, false));
        }

        public override string ToString()
     {
        //todo: implement the return string format
        return "(" + Event + "->" + Process.ToString() + ")";
     }
}
```

Appendix F: Composition.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using PAT.AMC.Assertions;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.Common.Classes.Ultility;
using PAT.Common.Classes.Assertion;
using PAT.Common.Classes.ModuleInterface;

namespace PAT.AMC.LTS
{
   public class Composition: Process
   {
      public Process FirstProcess;
      public Process SecondProcess;

      public Composition(Process first, Process second)
      {
         FirstProcess = first;
         SecondProcess = second;
         ProcessID = FirstProcess.ToString() + "|" + SecondProcess.ToString();
         ProcessID = DataStore.DataManager.InitializeProcessID(ProcessID);
      }

      public override string ToString()
      {
         return FirstProcess.ToString() + "|" + SecondProcess.ToString();
      }
      public override void MoveOneStep(Valuation GlobalEnv, List<Configuration> list)
      {
         System.Diagnostics.Debug.Assert(list.Count == 0);

         bool allTerminationCount = true;
         bool hasAtomicTermination = false;

         for (int i = 0; i < 2; i++)
         {
            Process process;

            if (i == 0)
               process = FirstProcess;
            else
            {
               process = SecondProcess;
            }

            List<Configuration> list1 = new List<Configuration>();
```

```
process.MoveOneStep(GlobalEnv, list1);
bool hasTermination = false;

for (int j = 0; j < list1.Count; j++)
{
   Configuration step = list1[j];

   if (step.Event == Constants.TERMINATION)
   {
      hasTermination = true;

      if (step.IsAtomic)
      {
         hasAtomicTermination = true;
      }
   }
   else
   {
      if (AssertionBase.CalculateParticipatingProcess)
      {
         step.ParticipatingProcesses = new string[] {i.ToString()};
      }

      Composition composition;

      if(i == 0)
      {
         composition = new Composition(step.Process, SecondProcess);
      }
      else
      {
         composition = new Composition(FirstProcess, step.Process);
      }

      step.Process = composition;
      list.Add(step);
   }
}

////to check whether there are synchoronous channel input/output
//if (Specification.HasSyncrhonousChannel)
//{
//    SynchronousChannelInputOutput(list, i, GlobalEnv, null);
//}

if (!hasTermination)
{
```

```
                allTerminationCount = false;
            }
        }

        if (allTerminationCount)
        {
            Configuration temp = new Configuration(new Stop(),
Constants.TERMINATION, GlobalEnv, false);

            if (hasAtomicTermination)
            {
                temp.IsAtomic = true;
            }

            if (AssertionBase.CalculateParticipatingProcess)
            {
                temp.ParticipatingProcesses = new string[2];
                for (int i = 0; i < 2; i++)
                {
                    temp.ParticipatingProcesses[i] = i.ToString();
                }
            }
            list.Add(temp);
        }

        //return returnList;
    }
        }
}
```

Appendix G: MoveInLocation.cs

```
using System.Collections.Generic;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.AMC.Assertions;

namespace PAT.AMC.LTS
{
  public class MoveInLocation : Process
  {
    public Process Process_Moved;
    public Location Location_Moved;
```

```
        public MoveInLocation(Location Loc,Process Pro)
        {
            Process_Moved = Pro;
            Location_Moved = Loc;
            ProcessID = "in." + Location_Moved.ToString() + "->" +
Process_Moved.ToString();
            Location_Moved.ProcessIncludeList.Add(Process_Moved);
            Process_Moved.LocationInclude = Location_Moved;
            ProcessID = DataStore.DataManager.InitializeProcessID(ProcessID);
        }


        public override string ToString()
        {
            return "(in." + Location_Moved.ToString()+" -> "+Process_Moved.ToString()
+")";
        }

            public override void MoveOneStep(Valuation GlobalEnv, List<Configuration>
list)
        {
            System.Diagnostics.Debug.Assert(list.Count == 0);
            Process_Moved.LocationInclude = Location_Moved;
            Location_Moved.ProcessIncludeList.Add(Process_Moved);
            list.Add(new Configuration(Process_Moved, Location_Moved.LocationID,
GlobalEnv, false));
        }
            }
}
```

Appendix H: MoveOutLocation.cs

```
using System.Collections.Generic;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.AMC.Assertions;

namespace PAT.AMC.LTS
{
    public class MoveOutLocation : Process
    {
        public Process Process_Moved;
        public Location Location_Moved;
```

```
        public MoveOutLocation(Location Loc,Process Pro)
        {
            Process_Moved = Pro;
            Location_Moved = Loc;
            ProcessID = "out." + Location_Moved.ToString() + "->" +
Process_Moved.ToString();
            Location_Moved.ProcessIncludeList.Remove(Process_Moved);
            Process_Moved.LocationInclude = null;
            ProcessID = DataStore.DataManager.InitializeProcessID(ProcessID);
        }


        public override string ToString()
        {
            return "(out." + Location_Moved.ToString()+" ->
"+Process_Moved.ToString()+")";
        }

            public override void MoveOneStep(Valuation GlobalEnv, List<Configuration>
list)
        {
            System.Diagnostics.Debug.Assert(list.Count == 0);
            Process_Moved.LocationInclude = null;
            Location_Moved.ProcessIncludeList.Remove(Process_Moved);
            list.Add(new Configuration(Process_Moved, Location_Moved.LocationID,
GlobalEnv, false));
        }
}}
```

Appendix I: OpenLocation.cs

```
using System.Collections.Generic;
using PAT.Common.Classes.Expressions;
using PAT.Common.Classes.Expressions.ExpressionClass;
using PAT.AMC.Assertions;

namespace PAT.AMC.LTS
{
   public class OpenLocation : Process
   {
      public Process Process_Moved;
      public Location Location_Open;


      public OpenLocation(Location Loc,Process Pro)
```

```
        {
            Process_Moved = Pro;
            Location_Open = Loc;
            ProcessID = "Open." + Location_Open.ToString()+ "->" +
Process_Moved.ToString();
            ProcessID = DataStore.DataManager.InitializeProcessID(ProcessID);
        }


        public override string ToString()
        {
            return "(Open." + Location_Open.ToString() + " -> " + Process_Moved.ToString()
+ ")";
        }

            public override void MoveOneStep(Valuation GlobalEnv, List<Configuration>
list)
        {
            System.Diagnostics.Debug.Assert(list.Count == 0);
            Process_Moved.LocationInclude = null;
            Location_Open = null;
            list.Add(new Configuration(Process_Moved, Location_Open.LocationID,
GlobalEnv, false));
        }
            }
}
```