Practical application of a light-weight formal implementation for specifying a multi-agent robotic system

Nadeem Akhtar[1] and Malik M. Saad Missen[2]

[1] ArchWare team, IRISA – University of South Brittany, Vannes, FRANCE
Assistant professor, The Department of Computer Science & IT, The Islamia University of Bahawalpur
Bahawalpur, 63100, PAKISTAN

[2] Assistant professor, The Department of Computer Science & IT, The Islamia University of Bahawalpur
Bahawalpur, 63100, PAKISTAN

Abstract

Light-weight formal specifications are flexible, have a concrete syntax, and play vital role in correctness of a multi-agent robotic system. To specify such systems in a way that it ensures correctness properties of safety and liveness is important, especially as these systems have high concurrency and in most of the cases have dynamic environment. We have considered a case-study of a multi-agent robotic system for the transport of stock between storehouses to exemplify light-weight formal specifications. The specifications have been modelled as a Labelled Transition System for light-weight formal verification..

Keywords: *Multi-Agent System, Formal methods, Light-weight formal methods, Finite State Process (FSP), Labelled Transition System (LTS), Safety property, Liveness property.*

1. Introduction

One of the most challenging tasks in software specification engineering for multi-agent robotic systems is to ensure correctness properties of safety and liveness, especially as these systems have high concurrency and in most cases have dynamic environment. Safety and liveness properties complete each other to ensure system correctness. Light-weight formal implementation of safety and liveness properties can play major role in system correctness.

Verifying that the code matches its requirement and design specifications is important. The understanding and expertise of formal methods requires time and technical expertise. Formal languages require mathematical training as a pre-requisite therefore they are less accessible to programmers. They use semantics that is very different from the semantics of the main-stream semi-formal programming languages. In most cases, formal specifications have operations based on complex mathematical concepts. Projects having formal specifications longer than the implementation code are simply too costly. The light-weight formal specifications are executable and practical. In some cases we can use light-weight formal language based specifications to only specify critical portions of the system, this can limit the number of states by system composition and hide the internal actions.

An agent is considered as a computer system situated in some environment, capable of autonomous actions in this environment in order to meet its design objectives [13]. Multiple agents are necessary to solve a problem, especially when it involves distributed data, knowledge, or control. A multi-agent system is a collection of several interacting agents in which each agent has incomplete information or capabilities for solving the problem [7]. These are complex systems and their specifications involve many levels of abstractions.

Our system consists of small robotic agents that work in a closed environment. LTS specifications based on FSP language have been used for specification definition of our multi-agent robotic system. In this work, requirement specifications are defined by using Gaia multi-agent method [14]. These Gaia specifications define early requirements, and then these specifications are transformed into light-weight formal LTS specifications for checking correctness properties of liveness and safety. These Gaia specifications are also used to define the system architecture specification by using π-ADL dot NET

specifications. Finally system is implemented by using a service-oriented architecture. Figure-1 shows our approach.

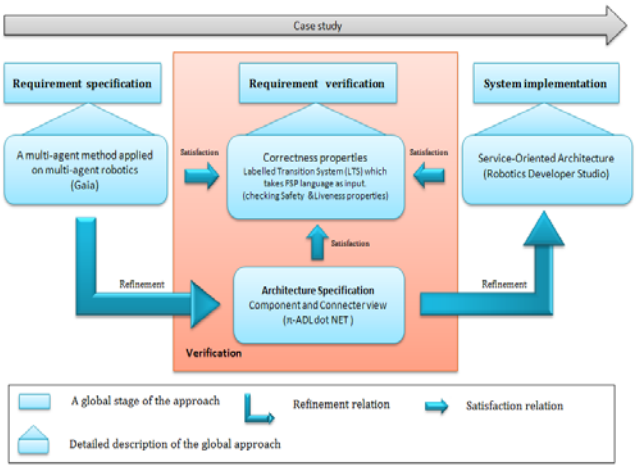


Fig.1 Approach and Proposed solution

This paper concentrates on the requirement verification portion of our approach. It put forwards the role of light-weight formal specifications for ensuring correctness properties of safety and liveness. The requirement specification phase, satisfaction relation from requirement specification to requirement verification, refinement relation from requirement specification to architecture specification, refinement relation from π-ADL dot NET architecture specification to service-oriented implementation, satisfaction relation from service-oriented implementation to requirement verification are not covered in this paper. These light-weight formal specifications are flexible, less rigorous and more practical than formal specifications and play vital role in ensuring correctness properties of safety and liveness. Therefore by using light-weight formal methods we are able to obtain a concurrent system in which there are processes working in parallel and there are synchronizations between different processes.

The labelled transition system and its associated analysis tool LTSA have an incremental and interactive approach to the development of component based systems. Consequently, components can be designed and debugged before composing them into larger systems. The goal is to specify our system by using light-weight formal FSP notation along with LTS to prove correctness properties.

2. Background studies

2.1 Light-weight Formal methods

Semi-formal methods do not give sufficient results in terms of precision and preciseness; they don't have the formal verification aspects. In contrast a formal specification method has precise mathematical semantics which in turn support formal verification.

We can prove system correctness by using formal methods. Here the word prove means mathematical rigorous proofs that specifications are according to the objectives, code is according to the specification, and code produces only the results that are required. These methods can achieve complete exhaustive coverage of the system thus ensuring that undetected failures in behavior are excluded. The core objective of a solid formal approach is to provide unambiguous and precise specification [4]. Correctness of the system is proved by formalizing the specifications of each component and process. Formal methods provide the formal analysis of the software. This formal analysis can be done manually, can be completely automated, or can be achieved by a combination of tools with human assistance. A formal method uses formal tools and notations; it uses mathematical notation consisting of set theory and logic but can also use a much more complicated notation. The requirements model based on mathematics create precise specification of the software, and ensures correctness. The formal representation of software requirements provides a way for logical reasoning about the construct produced and this achieves precise description and allows a stronger design that satisfies the required properties.

Formal notations are used to produce a complete detailed representation of the system that helps in the understanding, design, and development of the system. The requirements for distributed, large and complex systems are complicated, problematic at the initial stages and evolve periodically throughout the life cycle. This creates a need for the method of requirement implementation to be flexible and robust, so that it can easily accommodate the continuous versions of change [8]. This leads us to think about a light-weight implementation of formal methods. Formal methods can be applied to selected components of a system in varying degrees, depending on the needs. This degree of formalization can range from very formal to very informal. Light-weight formalization lies at a level between very formal and very informal. In very formal, meta-language is also formalized. In the very informal model, the meta-language can be a simple natural language representation. This range of

formalization gives us the flexibility to apply formal methods according to our requirements and implementation rigor. Light-weight formal methods will provide the advantages of formalism and at the same time will reduce drawbacks due to over formalization.

Model-checking [1] [3] is a type of formal method used to verify concurrency properties; it can be viewed as exhaustive investigation of a system state space to prove certain correctness properties. Process calculi based symbolic techniques such as π–ADL [11], CSP [6], CCS [10], ACP [2] and LOTOS [12] provide formal specifications for complex systems. Here complex means a system with a large number of independent interacting components, with concurrency between components and constant evolution. As formal verification techniques are getting more mature, our capability to build complex systems also increases.

2.2 Formalism: Labelled Transition System

LTS is a collection of techniques for the automated formal verification of finite-state concurrent systems. It consists of interacting finite state machines along with their properties; it performs compositional analysis to exhaustively search for violations of the required properties. Each component of a specification is described as LTS, which has all the possible states a component can reach and all possible transitions it can perform.

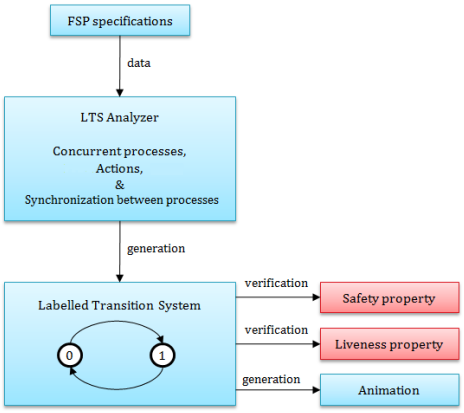


Fig.2 LTS Analyzer takes FSP as input

FSP is a process algebra notation having finite state processes used for the concise description of component behavior particularly for concurrent systems. It is a light-weight implementation of formal methods that provides construct to formalize specifications of software components and architecture. Each component consists of processes; each process has a finite number of states and is composed of one or more actions. There exists concurrency between elementary calculator activities for which there is a need to manage the interactions, communication and synchronization between processes. [9] proposed an analysis tool LTS Analyzer for FSP notation.

2.3 Correctness: Safety and liveness properties

Safety property is an invariant which asserts that "something bad does never happen", that is an acceptable state of the system is maintained. For example, a property which assures that a power reactor temperature would never exceed 100 degree Centigrade etc. [9] have defined safety property S = {a1, a2 … an} as a deterministic process that states that a trace consisting of the actions in the alphabet of S, is accepted by S. ERROR conditions are like exceptions which present the states that are not required. In complex systems safety properties are specified by directly specifying what is required.

Liveness property states the "something good happens" that shows and specifies the states of system that can be brought about by an agent under certain given conditions [14]. Progress is a type of liveness property. Progress P = {a1, a2 ... an} defines a property P which states that at least one of the actions from a1, a2 … an will be executed infinitely in an infinite execution of the system. [5].

3. CASE STUDY: Multi-Agent Robotics Transport System

In this section we present a case study of multi-agent robotics system. It is a system composed of robotic transporting agents. The objective is to specify our system and then verify the correctness properties of safety and liveness. The mission is to transport stock from one storehouse to another. They move in their environment which in this case is static i.e. topology of the system does not evolve at run time. There is a possibility of collision between agents during the transportation. Collision resolution techniques are applied to avoid system deadlocks. We have specified each and every part of the system i.e. agents along with the environment in the form of LTS.

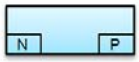
3.1 Types of Agents

There are three types of agents

1) *Carrier agent*: It transports stock from one store-house to another; can be loaded or unloaded and; can move both forward and backward direction. Each road section is marked by a sign number and the carrier agent can read this number.
2) *Loader / Un-loader agent*: It receives/delivers stock from the storehouse, can detect if a carrier is waiting (for loading or unloading) by reading the presence sensor, it ensures that the carrier waiting to be loaded is loaded and the carrier waiting to be unloaded is unloaded.
3) *Store-manager agent*: manages the stock count in the storehouse and it also transports the stock between storehouse and loader/un-loader.

3.2 Environment

There is a road between storehouse-A and storehouse-B which is composed of a sequence of interconnected sections of fixed length. Each road section has a numbered sign, which is readable by carrier agents. There are three types of road sections depending upon the topology of the road as shown in fig.3. Each of the three types of road sections has a unique numbered sign. The road is single lane and there is a possibility of collision between agents. There is a roundabout at storehouse-A and storehouse-B.



(a) N is the unique numbered sign. P is the parking Flag (TRUE or FALSE) e.g. the section that can be used as a parking.

(b) N is the unique numbered sign.

(c) Road section present at the loader and un-loader which detect the presence of the carrier agents at the loader and un-loader.

Fig.3 Environment consisting of road partitions

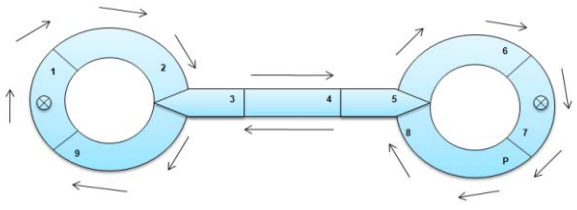
3.3 Scenario

In this case study we have used a road topology consisting of nine road partitions to represent all states and processes as shown in figure-3. It is the smallest circuit (i.e. combination of road partitions) that allows us to study all properties that would be in a much larger circuit. We have considered the case in which initially storehouse-A is full and storehouse-B is empty. The carrier task is to transport stock from storehouse-A to store-house-B until the storehouse A is empty. Loader at the storehouse-A loads, and the un-loader at the store-house-B unloads the carrier agent. The store-manager keeps a count of stock in each storehouse. In this case the environment is static. At the central section (3, 4, 5) there is a possibility of collision between carrier agents coming from the opposite directions. Priority is given to the loaded carriers i.e. if there is a collision between a loaded and an empty carrier than the empty carrier moves back and waits at the parking region during which the loaded carrier passes and unloads. The parking region as shown in the fig.3 consists of the road partition 8.

4. Light-weight LTS specifications and verification

4.1 Road – System Environment

In our case study the road is environment and each carrier has its particular route. The route is the path taken by carrier agents on the road to transfer stock from one storehouse to another. The route has been classified in two types the FULL_ROUTE path taken by loaded carriers and the EMPTY_ROUTE path taken by the empty carriers. The carrier agents move on the route in a clockwise direction. Here below are the FSP specifications for the route.



The LTS generated by these formal FSP specifications is shown in fig.4.

```
range R = 1..9
ROUTE = EMPTY_ROUTE[9],
FULL_ROUTE[v:R]=(
  when (v==7)    readunloadSign -> FULL_ROUTE[v]
| when (v!=7)    readSign[v]    -> FULL_ROUTE[v]
| when (v>=1&v<=6)movetonext    -> FULL_ROUTE[v+1]
| when (v==7)  waitforunloading -> EMPTY_ROUTE[7]
),
EMPTY_ROUTE[v:R]=(
  when (v==1)  readloadSign   -> EMPTY_ROUTE[1]
| when (v!=1)  readSign[v]    -> EMPTY_ROUTE[v]
| when (v==7)  movetonext     -> EMPTY_ROUTE[v+1]
| when (v==8)  movetonext     -> EMPTY_ROUTE[5]
| when (v==5)  movetonext     -> EMPTY_ROUTE[v-1]
| when (v==4)  movetonext     -> EMPTY_ROUTE[v-1]
| when (v==3)  movetonext     -> EMPTY_ROUTE[9]
| when (v==9)  movetonext     -> EMPTY_ROUTE[1]
| when (v==3)  movetoprevious -> EMPTY_ROUTE[v+1]
| when (v==4)  movetoprevious -> EMPTY_ROUTE[v+1]
| when (v==5)  movetoprevious -> EMPTY_ROUTE[8]
| when (v==1)  waitforloading -> FULL_ROUTE[1]
).
```

4.2 Carrier agent

The next step is to specify the carrier agents i.e. specify the empty-carrier and full-carrier agents. Here only one carrier agent (e.g. c1) is taken to represent all the possible states of the system that can arise.

```
range R = 1..9
CARRIER = MOVE_EMPTY,
MOVE_EMPTY = (    readSign[s:R] -> {movetonext,movetoprevious} -> MOVE_EMPTY
                | readloadSign  -> waitforloading   -> MOVE_FULL
  ),
MOVE_FULL = (    readSign[s:R]  -> movetonext       -> MOVE_FULL
               | readunloadSign -> waitforunloading -> MOVE_EMPTY
  ).
```

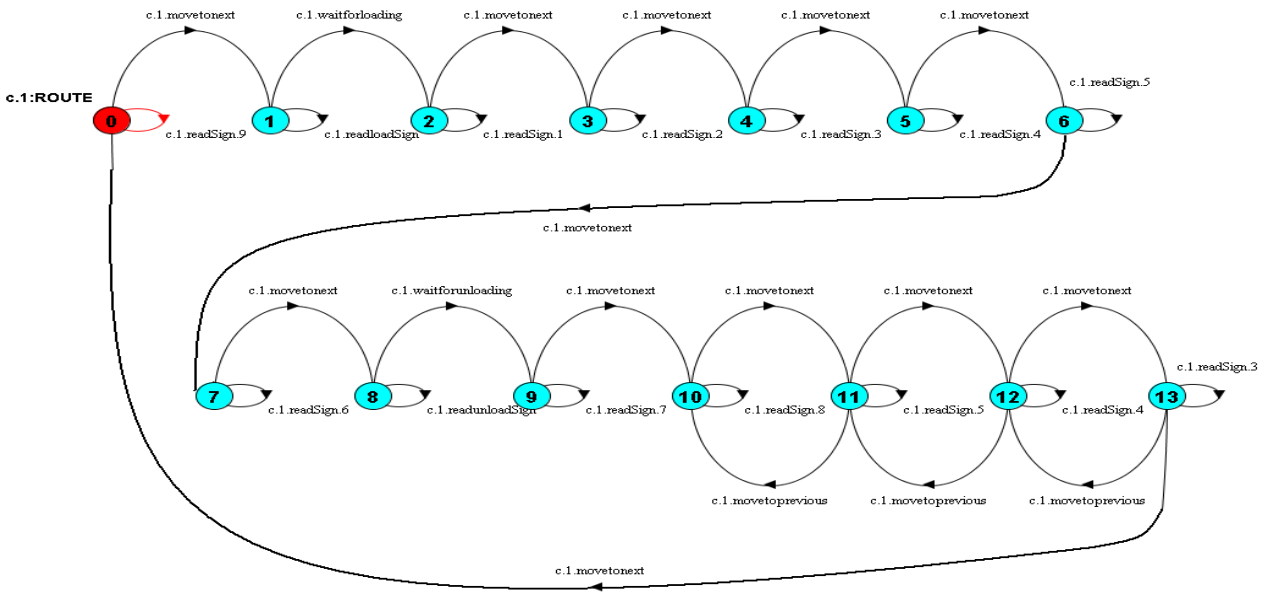


Fig.4 LTS specifications of the route (environment)

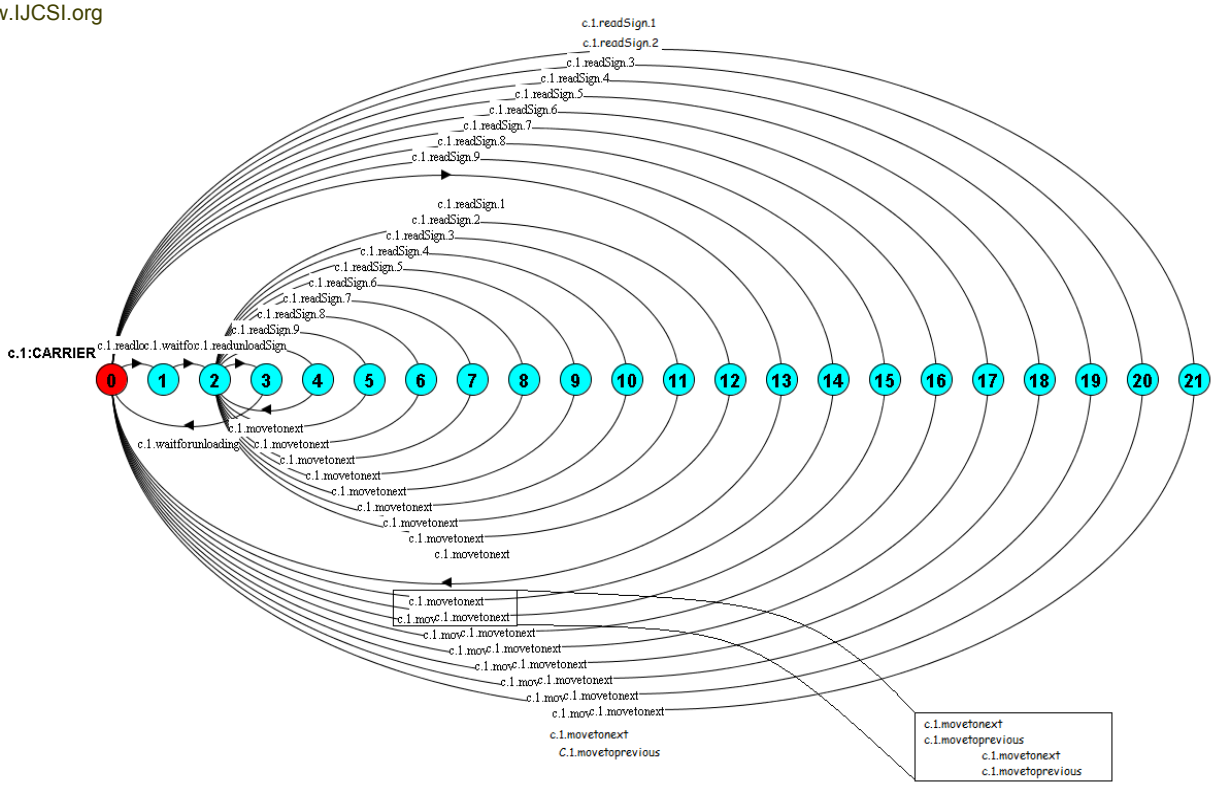


Fig.5 LTS specifications of Carrier agent

4.3 Loader & Un-loader agents

Loader and un-loader agent loads and un-loads the carrier agents respectively

```
1  LOADER = (waitfordeliver -> waitforloading -> LOADER).
```

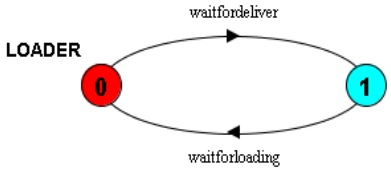


Fig.6 LTS specifications of Loader agent

```
1  UN_LOADER =
2  waitforunloading -> waitforreceive -> UN_LOADER).
```

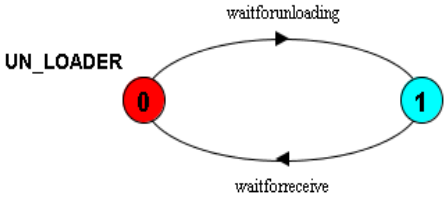


Fig.7 LTS specifications of Un-Loader agent

4.4 Stock management

Stock management ensures that the stock at the beginning of the case study at storehouse A is equal to the stock at the end of the case study at storehouse B.

```
1  const MaxS = 2            /// maximum number of Stock
2  range S = 0..MaxS
3
4  STOCKFULL_MANAGEMENT = STOCK_FULL[MaxS],
5  STOCK_FULL[st:S] =
6     ( stockCountA[st] -> STOCK_FULL[st]
7  | when(st>0)
8        decrementStockA -> send -> STOCK_FULL[st-1]
9  | when(st==0) stockEmptyA -> STOP).
10
11 STOCKEMPTY_MANAGEMENT = STOCK_EMPTY[0],
12 STOCK_EMPTY[st:S] =
13   ( stockCountB[st] -> STOCK_EMPTY[st]
14 | when(st<MaxS)
15       receive -> incrementStockB -> STOCK_EMPTY[st+1]
16 | when(st>=MaxS) stockFullB -> STOP).
17
18 ||STOCKSYSTEM =
19  (STOCKFULL_MANAGEMENT || STOCKEMPTY_MANAGEMENT)
20 /{decrementStockA/receive, incrementStockB/send}.
```

4.5 NOLOSS property

Safety property NOLOSS of Carrier agent infers that there is no loss of stock during the carrier load, unload, and movements between the storehouses. To represent the LTS here with all its states, we have taken a mini-route with only three road partitions. The carrier is loaded and then

the carrier is full, there is no loss of stock during the carrier agent's trajectory between storehouse A and B.

Safety property specifies the set of all traces that satisfy the property for a particular action alphabet. When the model produces traces, which are not accepted by the property automata then a violation is detected during reach ability analysis.

```
1   const N=2   // Number of carrier agents
2   const Min=0 // First(Load) road partition
3   const Max=3 // Last(Unload) road partition
4
5   property NOLOSS_Stock =
6      (empty.loaded -> ONTHEWAY[1]),
7
8   ONTHEWAY[part:Min..Max] = (
9      when(part>Min && part<Max)
10         full.moveto[part] -> ONTHEWAY[part+1]
11    | when(part==Max)
12         full.unloaded -> NOLOSS_Stock).
13  ||NOLOSS = (c[1..N]:NOLOSS_Stock).
```

5. Concluding remarks

The complete system is specified as a parallel composition of all processes and each process synchronize by means of shared actions. This paper focuses on the role of light-weight formal specifications for system correctness.

The development also depends upon the degree of formalism, the more the degree of formalism, the more the cost is, as formalism needs time, expertise, and human resources.

This approach has models based on formal methods and it revolves around lightweight formal verification of correctness properties (i.e. safety and liveness) in each phase from early requirements to the implementation i.e. Gaia multi-agent method requirement specifications, FSP-based LTS verification specifications.

The objective is to specify a multi-agent robotic system based on light-weight formal methods that are practical and feasible. The multi-agent robotic systems have con-currency, synchronization, correctness, and deadlock issues to be handled and formal light-weight development methods offer solutions for these issues. Another objective is the use of behavior analysis during analysis and design to discover correctness and safety problems early in the development cycle.

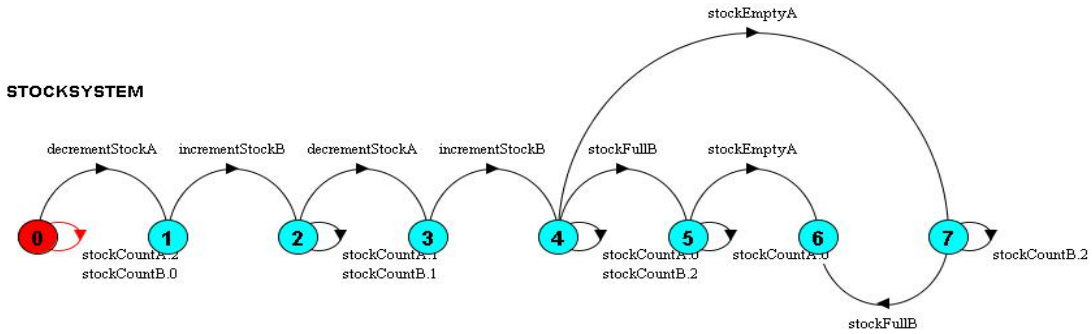


Fig.8 LTS specifications of stock management

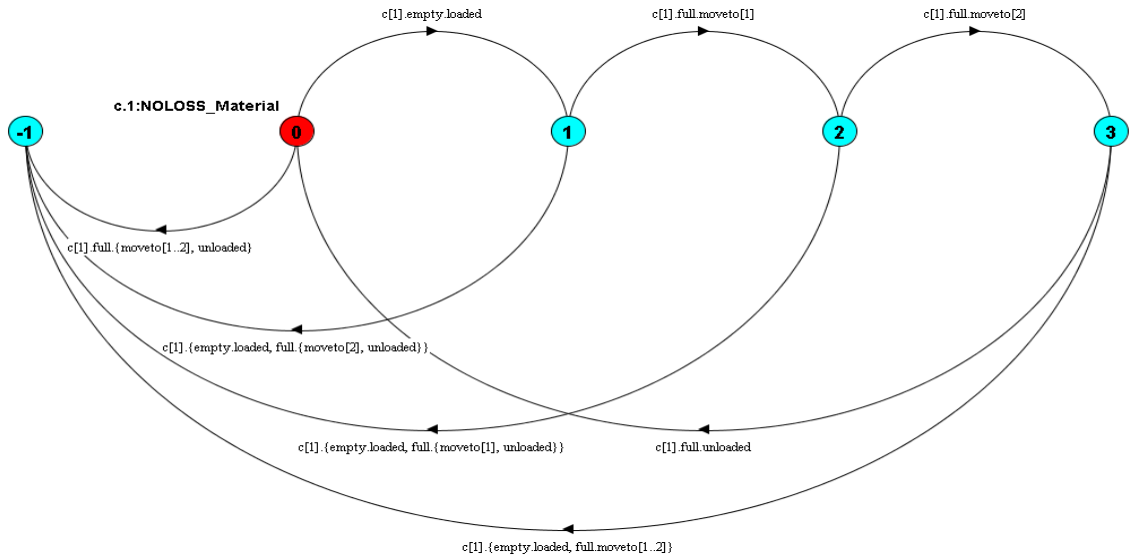


Fig.9 LTS specifications NOLOSS

Acknowledgments

We are grateful to Dr. Muhammad Mukhtar, *Vice Chanceller*, The Islamia University of Bahawalpur for his support for computer science and technology.

We are thankful to Prof. Dr. Flavio Oquendo for his continuous supervision and support. He is a Professor of Computer Science and Software Engineering at the University of South Brittany, part of the European University of Brittany, France, where he leads the ARCHLOG research team on Software Architecture.

We are grateful to Dr. Yann Le-Guyadec, Associate Professor of Computer Science and Software Engineering at the University of South Brittany, part of the European University of Brittany, France.

References

[1] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P,. and McKenzie, P. "Systems and Software Verification: Model-Checking Techniques and Tools". *Springer-Verlag*, 2001.

[2] Bergstra, J.A., and Klop, J.W. "ACPτ:A Universal Axiom System for Process Specification", *CWI Quarterly 15*, pp.3-23, 1987.

[3] Clarke, E., Grumberg, O., and Peled, D. "Model Checking". *MIT Press*, 2000.

[4] George, V., Vaughn, R. "Application of Lightweight Formal Methods in Requirement Engineering". *CROSSTALK: The Journal of Defence Software Engineering*, 2003.

[5] Giannakopoulou, D., Magee, J., and Kramer, J. "Fairness and priority in progress property analysis". *Technical report*, Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queens Gate, London SW7 2BZ, UK, 1999.

[6] Hoare, C. A. R. "Communicating sequential processes". *Communications of the ACM*, v.21 n.8, p.666–677, 1978.

[7] Jennings, N., Sycara, K., and Wooldridge, M. "A roadmap of agent research and development". *Int. Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7-38, 1978.

[8] Luqi and Goguen, J. "Formal Methods: Problems and Promises." *IEEE Software*, Volume 14, No 1, pp 73-85, 1997.

[9] Magee, J., and Kramer, J. "Concurrency: State Models and Java Programs". *John Wiley and Sons*, 2nd edition, 2006.

[10] Milner, R. "A Calculus of Communicating Systems", *Springer Verlag*, ISBN 0-387-10235-3, 1980.

[11] Oquendo, F. "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures", *ACM SIGSOFT Software Engineering Notes*, v.29 n.3, May 2004.

[12] Van Eijk, P.H.J. et al. "The Formal Description Technique LOTOS", *North-Holland*, Amsterdam, 1989.

[13] Wooldridge, M., and Jennings, N. "Intelligent agents: Theory and practice". *Knowledge Engineering Review*, 10(2):115-152, 1995.

[14] Zambonelli, F., Jennings, N., and Wooldridge, M. "Developing multiagent systems: The gaia methodology".

ACM Transactions on Software Engineering and Methodology, 12(3):317-370, 2003.

Dr. Nadeem Akhtar is Assistant Professor at the Department of Computer Science & IT, The Islamia University of Bahawalpur. He completed his PhD from the research Lab. VALORIA of Computer Science, University of South Brittany (UBS), France in 2010. His research areas are formal specification, formal architecture, and service-oriented architecture for robotics.

Dr. Malik M. Saad Missen is Assistant Professor at the Islamia University of Bahawalpur. He completed his PhD from University of Toulouse in 2011. His research interests include text data mining, information retrieval, social network research. He is currently exploring formal specification fronts.