American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-5, Issue-7, pp-269-274
www.ajer.org
Research Paper Open Access

# From Verification to Implementation: UPPAAL to C++

Sidra Sultana[1], Fahim Arif[2]
[1](Department of Computer Software Engineering, MCS/ NUST, Pakistan)
[2](Department of Computer Software Engineering, MCS/ NUST, Pakistan)

***ABSTRACT:*** *Validation and Verification of safety critical systems is crucial and if done incorrectly can result in fatal loss. The research contribution is focused on providing the transformation mechanism from software verification to source code phase of software development life cycle. Modeling of the critical systems initializes with the formalism of requirements followed by early model verification. The verified model can be automated to get the high level language code via code generator. Basic steps of transformation starts with UPPAAL timed automaton as an input, then getting the XML structure of the automaton. On the basis of XML structure parse tree is generated to visualize the data structure to be used for the C++ source code generation. Finally the verification, kernel and elapsed time used by the safety, liveness, reachability, deadlock freeness properties and fairness property is presented. In real time systems, safety and deadlock freeness properties are among the most crucial verification properties because if the system is not safe then it leads to insecurities related to life, money, reputation and time. If the system is in deadlock state then the system is simply of no use. Thus verification of safety and deadlock freeness properties is mandatory as per the statistical report provided in the research.*

***Keywords:*** *Real time Systems, Formal properties, UPPAAL Model Checker, Properties Verification, Automation, C++ Source Code, XML*

## I. INTRODUCTION

In software development life cycle, there are generally five phases including Software Requirement Engineering, Software Design and Architecture, Implementation, Testing and Maintenance. In the field of software engineering, systems with safety concerns require accuracy and precision in the design. Cases like Therac-25 radiation overdosing (1985-1987), AT&T Telephone network outage (1990), Pentium FDIV bug (1994) and Ariane-5 Crash (1996) are among some failures of design and testing errors that leads to faulty systems and resulted in fatal loses relating human lives, money and organization's reputation [1].

In embedded software system, extensive modeling, simulation and verification is required and formal methods are used for the verification of the critical requirements like safety, utility, liveness, deadlock freeness, fairness etc. There are numerous applications of system modeling and verification in the aspects of bug detection, safety and analysis. Real time systems are one of the important type of embedded systems [2]. Depending on the nature of time element, real time systems are either discrete or continuous. Certain mathematical models, graph theories and axioms are used for the verification of the real time systems.

## II. RELATED WORK

In literature, there are transformation rules for automation of source code to an automaton and vice versa. A new model for verification of Chapel programs is defined by T. K. Zirkel [1]. The spawning of the threads and the parallel constructs in Chapel for arbitrary scope is mapped to the model in a natural way. Feasibility of defect-detection and automatic verification is being demonstrated in the symbolic execution using the model checking for non-trivial Chapel programs. Chapel language is an extensive language where Chapel Verification Tool (CVT) is a prototype tool which is initially composed for small sets of code. Handling the arbitrary domains and complex data-types are to be provided in the extended version of tool covering even more portion of the Chapel Language. Partial Order techniques are being used to improve the scalability of the tool. On the basis of activity and sequence diagrams, researchers [2] have presented an automated methodology for transformation while preserving the system's object oriented view in consideration. Automatic discovery of deadlocks is being facilitated by mCRL2 tool set and to prove the formal properties, temporal logic is required for the application-specific properties to be verified in model checking tool. Quantitative information like number of requests, resource usage and expected execution time are modeled as annotations while profiling in

UML diagrams [4]. Reliability and efficiency of the system is accessed via these profiled and logged quantities. In order to enhance the formalism and usage of modeling, some well integrated tools like CADP are available. Bouissou [9] presented the converted code of hybrid automaton from the control code. In order to preserve the transformation, he provides the semantics from H-Simple to a sampled hybrid automaton. Similar to the annotations provided in [3], [9] uses statements for controlling the actuators and sensors along with starting from the system's control code.

## III. TRANSFORMATION RULES

Timed-automaton modeled in UPPAAL is used in software design and software testing phase. Thus, translation rules can be applied for converting UPPAAL model into source code for transition from software design phase into the implementation phase or for the sake of automating the code generation once the model is provided. Same UPPAAL model with verification properties can be used for reverse engineering the verification phase back to the implementation phase. Model Checking is a promising approach to ensure the safety of life critical systems but at the same time multiplying the efforts in terms of manually modeling the automaton of the system and then to verify the safety properties. This research contribution is focused on the automation of UPPAAL automaton into C++ Code. Input, transformation rules and output of this automation process is described as follows:

***Input***: The system that is being modeled in the UPPAAL Model Checker has two main parts which are Editor and Verifier. In the Editor's interface, the system is modeled in the form of automaton as shown in figures below:

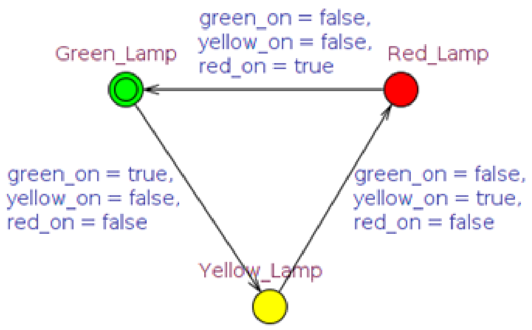


**Fig. 1** Lamp Model

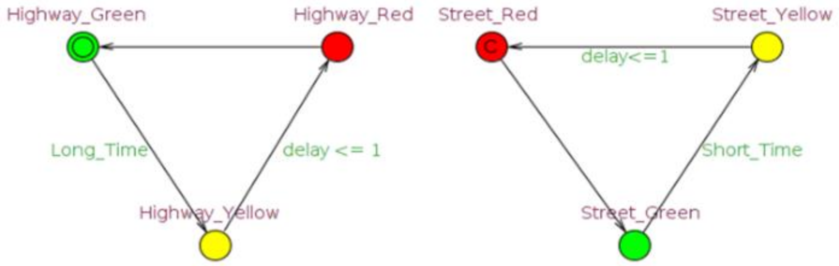


**Fig. 2** Timer Model

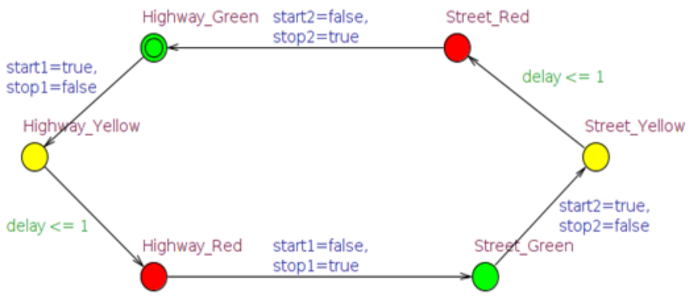


**Fig. 3** Controller Model

In the verifier section, properties are mentioned in TCTL logic as follows:
Safety Property: Traffic flow in highway implies that there is no traffic flow in street and thus the system is safe i.e., A[] ( Controller.start1 == true imply Controller.start2==false )
Reachability Property: If start1== true and stop1==false (vehicles move in Highway path) i.e., E<> (Controller.start1==true & Controller.stop1==false)
Deadlock Freeness: In this system, any deadlock was not occurred i.e., A[] not deadlock
Liveness Property: In this rule, if start1==true and stop1==false ( vehicles move in Highway) or start2==true and stop2==false (vehicles move in Street), then the green light should on i.e., E<> (Controller.start1==true & Controller.stop1==false | Controller.start2==true & Controller.stop2==false ) > (Lamp.green_on== true )
Fairness: In all of the states, delay <= 1 variable is const i.e., A<> Controller.delay<=1

***Transformation Rules:*** Transformation rules relates to semantics of parsing xml tags and converting them into C++ constructs like structures, variables, expressions, conditional operators, loops and functions. Fig. 4 shows basic phases of transformation from timed automaton xml to C++ program.

Parse tree generation is the second phase in which the basic structure types are at one level and nested elements are parsed as their child nodes. Fig. 5 shows the XML Parse tree.
***Output***: After applying the transformation rules, this automaton and xml file of UPPAAL is to be transformed in C++ code. Almost all of the data structures of C++ are either derived from Array or Structures. The automaton generated from UPPAAL resembles with the Graphical Data Structure. Graphs in C++ are either implemented with Adjacency Matrix (two dimensional arrays of locations (nodes) and transitions (edges)) or as Adjacency list. First approach is listed in Table 1 and Table 2 below:

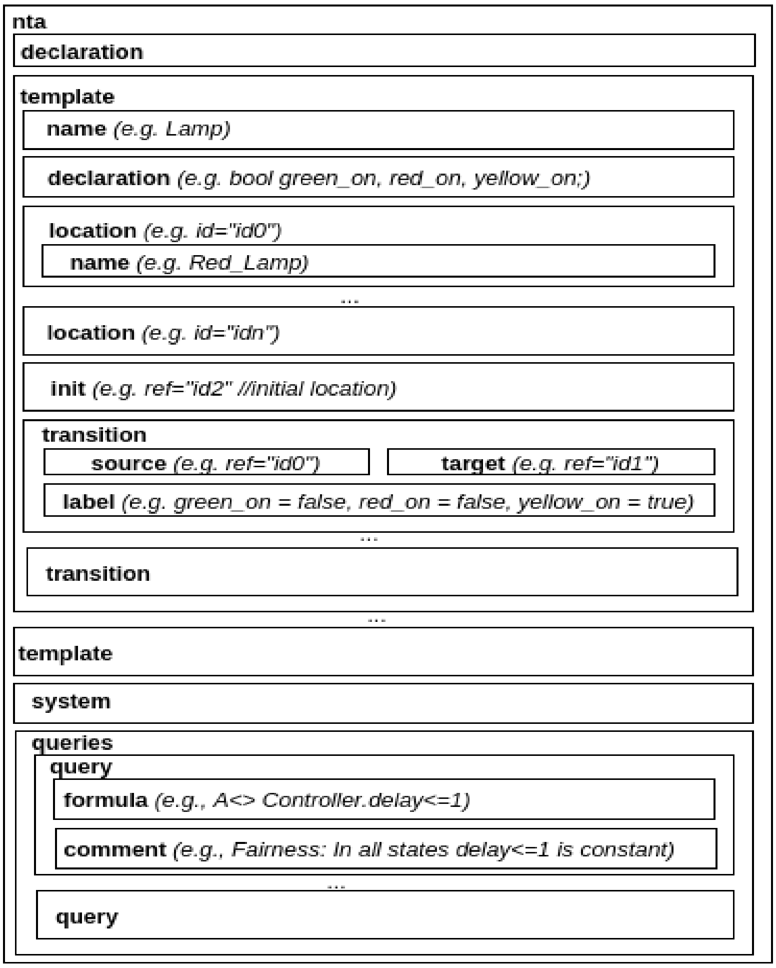


**Fig. 4** XML Structuring

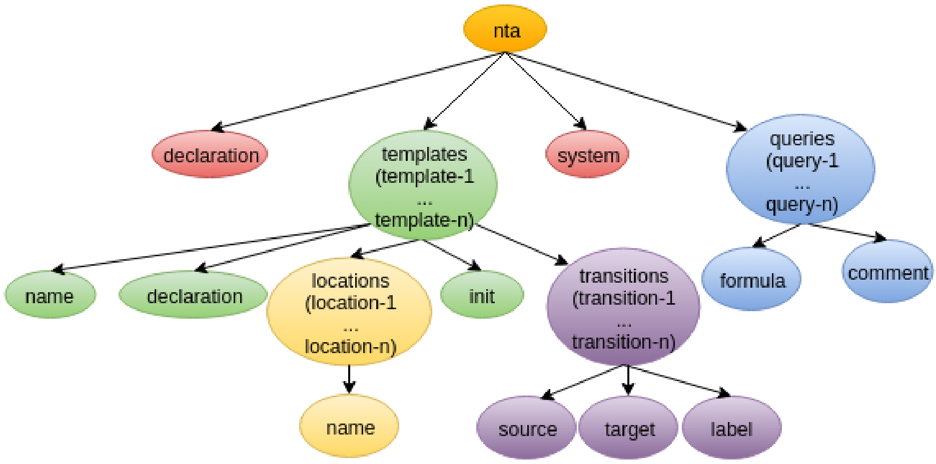


**Fig. 5** XML Parse Tree

**Table 1** Adjacency Matrix for Locations (nodes)

| Id | x | y | Label Title | Label x | Label y | color | Initial state |
|---|---|---|---|---|---|---|---|
| **id2** | -153 | -85 | Green Lamp | -204 | -119 | #00ff00 | true |
| **id1** | 42 | -85 | Red Lamp | 32 | -119 | #ff0000 | false |
| **id0** | -51 | 68 | Yellow Lamp | -102 | 34 | #ffff00 | false |

**Table 2** Adjacency Matrix for Transitions (edges)

| **Target** | | | |
|---|---|---|---|
| **Transitions** | **Green_Lamp** | **Red_Lamp** | **Yellow_Lamp** |
| **Green Lamp** | N/A | N/A | green_on = true, yellow_on = false, red_on = false |
| **Red Lamp** | green_on = false, yellow_on = false, red_on = true | N/A | N/A |
| **Yellow Lamp** | N/A | green_on = false, yellow_on = true, red_on = false | N/A |

The adjacency list can be created with either node or transition perspective. Node structure of adjacency list construct is as follows

```
struct node{
    char id; //id2
    int x; //-153
    int y; //-85
    char lable_title; //Green_lamp
    int lable_x; //-204
    int lable_y; //-119
    int color; //#00ff00
    bool initial; //true
    node *next;
};
```

Transition structure of adjacency list construct is as follows

```
struct transition{
char source; //id1
    char target; //id2
    char label_kind; //assignment, update, guard or reset
    int lable_x; //-102
    int lable_y; //-144
    char label_text; //green_on=false, yellow_on= false, red_on = true
    transition *next;
```

};
The structure for the node and transition is customized and the functions of the standard linked list will be used in transformation rule set.

## IV. DISCUSSION AND ANALYSIS

Core objective of the research is to transform the timed automaton for UPPAAL model checker into C++ source code. UPPAAL requires a timed automaton for modeling purpose and control temporal logic (CTL) for verification purpose. Transformation rules are focusing the modeling part of UPPAAL model checker and intermediate artifacts like xml structure and parse tree are generated to get some visual data structure that can be further transformed in the code generation process. For efficient verification of the safety critical properties accurate response time is mandatory. Table 3 shows the verification, kernel and elapsed time (in seconds) used in the verification of safety, reachability, deadlock, liveness and fairness properties. Graphical representation of time used by the verification properties is presented in Fig. 6.

**Table 3** Time used for the execution of verification properties

| Properties | Verification Time Used | Kernel Time Used | Elapsed Time Used |
|---|---|---|---|
| **Safety** | 0.01 | 0 | 0.02 |
| **Reachability** | 0 | 0 | 0.001 |
| **Deadlock** | 0.01 | 0.01 | 0.018 |
| **Liveness** | 0 | 0 | 0.001 |
| **Fairness** | 0 | 0 | 0.002 |

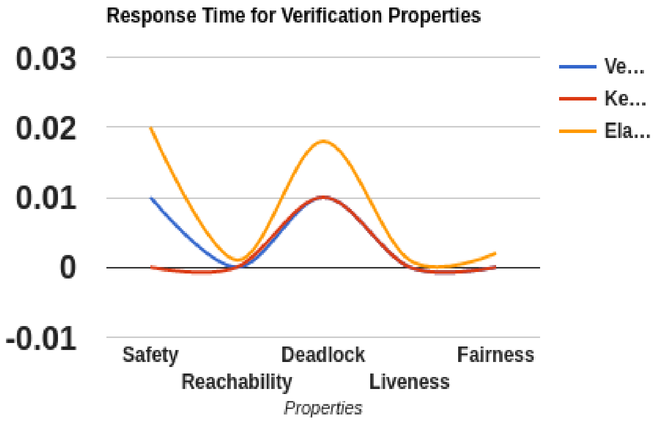


**Fig. 6** Response Time for Verification properties

Verification time used by the UPPAAL model checker is worth mentioning for Safety and Deadlock freeness properties. Kernel responded all properties in no time but the deadlock property took 0.01seconds. Elapsed time is the one that responded uniquely for all of the verified properties. Safety property took maximum elapsed time i.e., 0.02 seconds and with fractional change deadlock freeness property has 0.018 seconds of elapsed time used.

## V. CONCLUSION

In real time systems, testing phase is important in terms of ensuring critical properties like safety, utility, deadlock freeness, reachability, fairness, mutual exclusion, liveness etc. Such highly complex systems cannot affords compromising the fine grain safety concerns specially in the fields of fabrication, chip design, transportation, medical, satellite etc. Modeling of the critical systems initializes with formalism of requirements followed by early model verification. The verified model can be automated to get the high level language code via code generator. Software testing for real time systems is generally decomposed in modeling and verification phases. Timed-automaton modeled in UPPAAL is used in software design and software testing phase. Thus, translation rules can be applied for converting UPPAAL model into source code for transition from software design phase into the implementation phase or for the sake of automating the code generation once the model is provided. Same UPPAAL model with verification properties can be used for reverse engineering the verification phase back to the implementation phase. This research contribution is being carried out to reverse engineer the UPPAAL automaton into C++ constructs.

## REFERENCES

[1] Kulkarni, V.; Tata Res. Dev. & Design Centre, Pune; Reddy, S.,: Introducing MDA in a large IT consultancy organization, IEEE,Software Engineering Conference, 2006. APSEC 2006. 13th AsiaPacific, 2006, 6-8 Dec. 2006, pp. 419 - 426.
[2] Choong Koon Fong.,: Successful Implementation of Model Driven Architecture, APAC Support Center, 2007, A Borland White Paper.
[3] Behzad Bordbar; Kyriakos Anastasakis, MDA and Analysis of Web Applications, School of Computer Science, University of Birmingham, Birmingham, 2007.
[4] YU Xiaofeng; HU Jun; ZHANG Yan et al, A Model Driven Development Framework for Enterprise Web Services, Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, EDOC, 2006, pp. 75-84
[5] Mila Keren; Andrei Kirshin; Julia Rubin; Ahto Truu, MDA Approach for Maintenance of Business Applications, Model Driven Architecture –Foundations and Applications, Lecture Notes in Computer Science Volume 4066, ECMDA-FA 2006, LNCS 4066, pp. 40 – 51
[6] Hong Wang ; Carleton Univ., Ottawa, Ont., Canada ; Dong Zhang ; Jun Zhou, MDA-based development of e-learning system, Computer Software and Applications Conference, 2003. COMPSAC 2003, pp. 684 - 689
[7] John D. Poole, Model-Driven Architecture: Vision, Standards And Emerging Technologies, ECOOP, 2001.
[8] Youcong Ni ; Wuhan, China ; Shi Ying ; Linlin Zhang ; Jing Wen , Modeling Aspect-Oriented Software Architecture, Industrial and Information Systems, International Conference on Industrial and Information Systems, 2009, IIS '09, pp. 108 - 113
[9] Jiang Guo, An approach for modeling and designing software architecture,Engineering of Computer-Based Systems, 2003. Proceedings. 10Th IEEE International Conference and Workshop, pp. 89 – 97
[10] Jiang Guo ; Yuehong Liao, The scheduling algorithms in software architecture modeling, Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop, pp. 36 – 43
[11] Ortega, D. ; Silvestre, L. ; Bastarrica, M.C. ; Ochoa, S.F., A Tool for Modeling Software Development Contexts in Small Software Organizations, Chilean Computer Science Society (SCCC), 2012 31st International Conference, pp. 29-35
[12] Chang-Guo Guo ; Xiao-Ling Li ; Jun Zhu., A Generic Model for Software Monitoring Techniques and Tools, Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference, pp. 61-64
[13] Alves Pereira, J. ; Souza, C. ; Figueiredo, E. ; Abilio, R. , Software Variability Management: An Exploratory Study with Two Feature Modeling Tools, Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium, pp. 20-29
[14] T. K. Zirkel, S. F. Siegel, and T. McClory, Automated Verification of Chapel Programs Using Model Checking and Symbolic Execution, NASA Formal Methods Lecture Notes in Computer Science Volume 7871, 2013, Pp 198-212
[15] D. Remenska, J. Templon, T. A.C. Willemse, P. Homburg, K. Verstoep, A. Casajus, and H. Bal, From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent System, NASA Formal Methods Lecture Notes in Computer Science Volume 7871, 2013, Pp 244-260.
[16] S. Lyde, M. Might, Extracting Hybrid Automata from Control Code, NASA Formal Methods Lecture Notes in Computer Science Volume 7871, 2013, Pp 447-452
[17] N. Decker, M. Leucker, and D. Thoma, jUnit RV —Adding Runtime Verification to jUnit, NASA Formal Methods Lecture Notes in Computer Science Volume 7871, 2013, Pp 459-463.
[18] Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
[19] Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP 2010: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Program- ming, pp. 51–62. ACM Press (2010)
[20] Cousot, P.: Integrating physical systems in the static analysis of embedded control software. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 135–138. Springer, Heidelberg (2005)
[21] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
[22] Bouissou, O.: From control-command synchronous programs to hybrid automata. In: Analysis and Design of Hybrid Systems, pp. 291–298 (2012)
[23] Najafi M., H. Haghighi: A formal mapping from Object-Z specification to C++ code, Scientia Iranica D (2013) 20(6), pp. 1953-1977
[24] Najafi M., Haghighi H., Zohdi N. T.: A survey on formal, object-oriented program development approaches, Scientia Iranica D (2015) 22(3), pp. 1001-1017
[25] M. Bashiri and S.G. Miremadi: Perform-ability guarantee for periodic tasks in real-time systems, Scientia Iranica D (2014) 21(6), pp. 2127-2137
[26] Ralf W., Nils J., Erika A., Joost-Pieter K.High-Level Counter examples for probabilistic automata, Logical Methods in Computer Science Vol. 11(1:15)2015, pp. 1–23
[27] Rahim, M.A.B., Arif, F.,: Translating Activity Diagram from Duration Calculus for Modeling of Real-Time Systems and its Formal Verification using UPPAAL and DiVinE, Mehran University Research Journal of Engineering & Technology, Volume 35, No. 1, January, 2016, pp. 139-154
[28] Rahim, M.A.B., Ahmad, J., and Arif, F.,: Parallel verification of UML using DiVinE tool, 5th International Conference on Computer Science and Information Technology, Amman, Jordan, 27-28 March, 2013, pp. 49-53