Automatic Generation of Systematic Test Cases Using AADL for Embedded Software

Chunyan Ma, Yue Li, Yunwei Dong, and Yaqi Liu

***Abstract*—AADL is used to design embedded software in ever-increasing mission-critical applications. With the complexity of embedded software increasing, integration testing and system testing based on codes are becoming more difficult. This paper describes a systematic test cases generation approach using AADL for embedded software. The approach uses hierarchical testing model to generate test cases which is fully automatic model-driven. This paper designs one set of mapping rules from AADL to hierarchical testing model for constructing it automatically. The case study shows experimental process of the test model construction and test case generation. Automatic generation of systematic test cases using AADL for Embedded Software is feasible.**

***Index Terms*—AADL; test cases generation; hierarchical testing model.**

I. Introduction

Model-Based Testing (MBT) is a technique that tries to address the problem of high-cost and low-efficiency testing by introducing automatic generation of tests from models representing the behaviour of the system. AADL [1] provides modelling concepts to describe the runtime architecture of application systems in terms of concurrent tasks and their interactions as well as their mapping onto an execution platform. According to the broad prospects of AADL, AADL-based testing will be one of the further explorations of AADL, which includes AADL-based test cases generation and model-based testing process, methods, tools, etc. [2], and one of the foundations of embedded systems successfully described in AADL. Few works investigate the testing technique for AADL. Cao [3] presents test cases automatic generation technique based on AADL model which includes three kinds of unit test. In our previous work [4], we proposed the test model of integration and system testing for AADL which is described by extended interface automaton, and the research focuses on the automatic construction algorithms from AADL design model to the test model. Through previous investigation, we find that the scale of the test model for complex system is growing too large, which leads to massive numbers of states. Accordingly, it is necessary to find a new description model to master the state explosion problem for integration and system testing. In this paper, the proposed hierarchical testing model (HTM) can cope with the state explosion problem encountered when applying state machine model to large software systems; besides, it can easily produce the system and integration test cases. HTM consists of several hierarchy levels which represent the whole system, the nested subsystem, process, thread, subprogram and their behaviours, interaction from the top to bottom. Our main goal is to use the topological and behavioural properties of a system described in AADL, as a foundation of MBT to generate the systematic test cases which are used for integration testing and system testing. Test model construction and test cases generation assistant tool is implemented by checking the input/output behaviour of the system and components described by AADL.

The remainder of this paper is structured as follows: Section 2 introduces AADL and interface automata. Formal definition of hierarchical testing model and mapping rules from AADL to hierarchical testing model are presented in Section 3. Section 4 describes the systematic test cases generation approach. Section 5 presents the test process and result obtained from case study. Section 6 concludes the paper by summarizing the work and suggesting future research directions.

Manuscript received April 10, 2012; revised May 5, 2012.

Chunyan Ma is with School of Software and Microelectronics, Northwestern Polytechnical University, Xi'an, China (e-mail: machunyan@nwpu.edu.cn).

Yue Li and Yunwei Dong are with School of Computer Science, Northwestern Polytechnical University, Xi'an, China

Yaqi Liu is with Institute of Computing Technology Chinese Academy of Sciences, Beijing, China

II. AADL and Interface Automata

AADL captures system structure by identifying architectural and behaviour components, communications through component port connections and components compositions. AADL components can be divided into different kinds of components: system level components, process level components and thread level components, along with the data components, subprogram components and execution platform components connected to them. Each component has one or more operating modes and mode change presenting some behavioural aspects. AADL is also an extensible language which can be used to define annexes. AADL behavioural annex specifies the detailed behaviour of threads and subprograms. In this paper, the topological and behavioural contents of a system including component feature, subcomponent, component behaviour specification and mode change, are extracted to construct HTM. As a result, the behavioural and functional high-level, platform-independent architecture is taken into consideration.

The paper captures the I/O behaviors of a component by an interface automaton. An interface automaton $P=<V_p, V_p^{init}, A^I_P, A^O_P, A^H_P, \triangle P>$ consists of the following elements[6]:

- V_p is a set of states.

$V_p^{init} \in V_p$ is a set of initial states. If $V_p^{init}=\emptyset$, then P is called empty.

A^I_P、A^O_P and A^H_P are mutually disjoint sets of input (?),

output(!), and internal actions(;). $A_P = A^I_P \cup A^O_P \cup A^H_P$ is denoted as the set of all actions.

$\triangle P \subseteq V_p \times A_P \times V_p$ is a set of steps. $\forall t \in \triangle P$, t is denoted as $t = <v_1, a, v_2>$, where $v_1 \in V_p$, $v_2 \in V_p$ and $a \in A_P$

III. Hierarchical Testing Model for AADL Model

Instead of taking a testing model and see how we can best exploit it for testing, let us consider how we should ideally propose and build a testing model so that the implementation based on AADL can be effectively tested. The proposed HTM for AADL in section 4.1 is in line with the testing target: (1) The HTM is automatically constructed from AADL according to the mapping rules given in section 4.2; (2) The HTM has components interaction and component behavior information for both system and integration testing to generate testing paths; (3) Test cases are automatically derived from the HTM. Comparing the HTM with our previous work [4], it has the following advantages: (1) more rich testing information; (2) state space is greatly reduced and more expressive.

A. Formal Definition of Hierarchical Testing Model

Definition 4.1 AADL hierarchical testing model described as HTM=< Ns, Ps, < Ns, Ps > , $\rightarrow^{link}$, $\rightarrow^{sub}$> consists of the following elements:

Ns = S∪subS∪Proc∪T∪subP is a set of nodes, where,

S is the root node which corresponds to the application system.

subS is a set of subsystem nodes which correspond to the system components of AADL as subsystems embedded in system.

Proc is a set of process nodes which corresponds to the processes of AADL.

T is a set of thread nodes which corresponds to the threads of AADL.

subP is a set of subprogram nodes which corresponds to the subprograms of AADL.

Ps is a set of interface automata which models the behaviors of the components.

$<Ns, Ps> \subseteq Ns \times Ps$ is a set of relations between node and interface automaton, meaning that the behavior of the node is described by an interface automaton, if n∈Ns, p∈Ps, p' ∈Ps, p ≠ p', then <n, p>∈<Ns, Ps> => <n, p'>∉<Ns, Ps>.

→sub ⊆ (Ps.Vp × Ns)∪ (Ns × Ns) is a set of subordinate relationships between one state node of interface automaton and one node in Ns or between nodes in Ns, where Ps.Vp denotes the state set of all interface automaton in Ps. If v∈Ps.Vp , <v, n>∈→sub，then

1) n≠S, p∈Ps;
2) v is a state of state set Vp in p;
3) n1∈Ns, <n1,p>∈<Ns,Ps>;
4)One of the following conditions is reached.
a) If n1 is S, then n∈subS;
b)If n1∈subS, then n∈Proc;
c)If n1∈Proc, then n∈T;
d)If n1∈T, then n∈subP.

The relationship between one state node of interface automaton and one node in Ns represents the connection between the state v_p and the underlying component. If $\forall <n_1, n_2> \in <Ns, Ns>$, then n_1 is the parent node of n_2.

$\rightarrow^{link} \subseteq Ns.Ps.A_P \times Ns.Ps.A_P$ is a set of links between interface automaton state transitions. $Ns.Ps.A_P$ denotes a set of actions of interface automata in Ps which has relations with one node in Ns. If (n.p.a, n.p'.a') $\in \rightarrow^{link}$, then <n,p>∈<Ns,Ps>, <n',p'>∈<Ns, Ps>, a∈A_P in p and a'∈A_P in p'. The link expresses the interaction between the subcomponents belonging to the same level.

HTM is divided into a hierarchy, where each part represents an AADL component node or the behaviour of AADL component. The component type contains system, process, thread and subprogram. At the same level, the interaction between different behaviours is modelled as "links", which means a transition in one interface automaton causing a transition in another interface automaton.

Mapping Rules from AADL to Hierarchical Testing Model

This section designs the mapping rules shown in Table I, II, and III from AADL to HTM. According to the mapping rules, section 6 develops the supporting tool which automatically builds the HTM by parsing the AADL model.

TABLE I: The Mapping Rules from AADL to HTM

AADL syntax
"system" component which is not declared in other system component
"system" component which is declared in other system component
"process" component
"Thread" component
"thread" group which includes n threads
"subprogram" component
"modes" of component A and its "subcomponents B" is active in mode m of modes.
"annex behavior_specification" of component A
If component A implementation has not contain the modes and behavior specification
"subcomponent B" of component A
"port connection" between component A and component B

TABLE II: FROM MODES OF COMPONENT TO INTERFACE AUTOMATON

Operational Modes Syntax [2]	Interface automaton p
initial mode	The initial state s0 of p
mode	One state s in V_p of p
an arrival single event associated with each mode transition	One element in $A^I_P \cup A^O_P$ of p
mode transition	One element in $\triangle$P of p

The key to automatically construct HTM from AADL model is what information should be extracted from the AADL model. The target of integration and system testing is to estimate the functional behaviour and how components interact through interfaces of components. In AADL standard, component is declared by component type and realized by component implement. A component type specifies the external interface of a component that its implementations would satisfy. A component implementation contains subcomponents and their connections, component property, component behaviour, and component modes. Table 3-1 describes the mapping rules from AADL to HTM. Component is transformed into the node in Ns of HTM by mapping rules 1-6. The behaviour specification and modes of a component are transformed by mapping rules 7-8. The details of mapping rule 7 and 8 are further elaborated by Table3-2 and Table3-3. The relationship of component and its subcomponent is mapping to the connection between the state in interface automaton of component and the subcomponent node with mapping rule 10. Mapping rule 11 presents how the interaction between components is transformed.

TABLE III: FROM BEHAVIOR SPECIFICATION OF COMPONENT TO INTERFACE AUTOMATON

Behavior specification syntax [4]	Interface automaton p
initial state	the initial state s0 of p
state	One state s in V_p of p
Events which the transition is guarded with	One element in A^I_P of p
Boolean conditions which the transition is guarded with	One element in A^H_P of p
"action?" which is attached to the transition	1) Add one new state which represents the state after the action. 2) mapping the action to one element in A^I_P of p
"action" which is attached to the transition	1) Add one new state which represents the state after the action. 2) mapping the action to one element in A^H_P of p
"action!" which is attached to the transition	1) Add one new state which represents the state after the action. 2) mapping the action to one element in A^O_P of p
state transition $<s_1,s_2>$ which denotes the source and target is respectively s_1 and s_2	If the action attached to the transition exist, and the added new state is called s, then mapping $<s_1,s_2>$ and $<s_2,s>$ to two elements in $\triangle$P of p

IV. TEST CASES GENERATION

In this section, we present our approach for test cases generation based on the HTM. Test cases generation from a Finite State Machine is a long-standing research problem with numerous contributions over decades. On this base, we give random path algorithm of test cases generation from HTM. Each test case is abstractly denoted by the pair sequences about the component node and the corresponding action. Let the test model HTM be < Ns, Ps, < Ns, Ps >, $\rightarrow^{link}$, $\rightarrow^{sub}$> , the form for each test case TC is: TC = $n_1.a_1$, $n_2.a_2$, …,$n_i.a_i$, where $n_k \in$ Ns, $p_k \in$ Ps, $a_k \in A_P$ of p_i ,$1 \le k \le i$. The random algorithm of each test cases generation is as follows.

Input: HTM=< Ns, Ps, < Ns, Ps >, $\rightarrow^{link}$, $\rightarrow^{sub}$>

Output: one test case TC

Algorithm description:

Initiate TC and Set the root node as the current component node n, n∈ Ns.

If an interface automaton p∈Ps which has relationship with the current node n exists, then continue the step 3, otherwise continue the step 6.

Set any initial state of an interface automaton as the current state node v, v∈ Vp in p.

If $\exists$ <v,a,v1>$\in \triangle$P in p or $\exists$ (v, m)$\in \rightarrow$sub , then execute 4.1, or 4.2, or 4.3 randomly

4.1 put the "n.a" into TC and set v1 as the current state node v.

4.2 if $\exists$<n.p.a, n.p'.a'>$\in \rightarrow^{link}$ and $\exists$ <v_2,a,v_3>$\in \triangle$P in p', then set v_3 of p' as the current state node v, v∈ V_p in p'.

4.3 if $\exists$ <v, m> $\in \rightarrow^{sub}$, m∈Ns , set m as the current component node n;

Loop step 4 until the condition of step4 is false. Then Continue step 7.

If <n, n_1> $\in \rightarrow^{sub}$, set n_1 as the current component node n. continue the step 2

End.

When an embedded system is being tested, the tester plays the role of the environment. An embedded system interacts closely with its environment by exchanging *input* and *output* signals. The tester needs to provide concrete value for each input of each abstract test case. Each abstract test case may derive many concrete test cases.

V. CASE STUDY

The paper develops the assistant tool prototype A2TC which can automatically parse AADL model into HTM and generated the test case set. The tool prototype A2TC and systematic test approach have been applied to code generated by three AADL cases which include cruise control system of the car, producer and consumer, and Flight System in [7]. In this section, the experimental process and results of cruise control system for the car is elaborated. Cruise control system simulates the driving process of the car. It includes two processes: CarSimulator and Controller. CarSimulator process simulates car driving which includes launch, stop, acceleration, deceleration etc. CarSimulator contains two threads: BasicThread and CarSimulatorThread. CarSimulatorThread runs only after the car started. Controller process is the cruise controller. When receiving the start signal 'on', Controller process executes the SpeedControl thread. SpeedControl sends periodical signal to BasicThread thread for controlling car speed.

Step1: the HTM generation: The tool A2TC parses AADL XML file and instance file of Cruise control system into the HTM represented by the Fig1. Due to limited space, the complete formal description for test model HTM is omitted. We take the graphical representation of test model to explain HTM. In Fig1, the CarSimulator node points to interface automaton mapped by the following modes specification of CarSimulator process.

```
modes
  engineoffstate: initial mode ;
  engineonstate: mode ;
  annex behavior_specification {**
  mode transitions
    engineoffstate -[ EngineOn? ]-> engineonstate;
    engineonstate -[ EngineOff? ]-> engineoffstate
{ throttle := 0.0; speed := 0; distance := 0; brakepedal := 0; };
```

**};

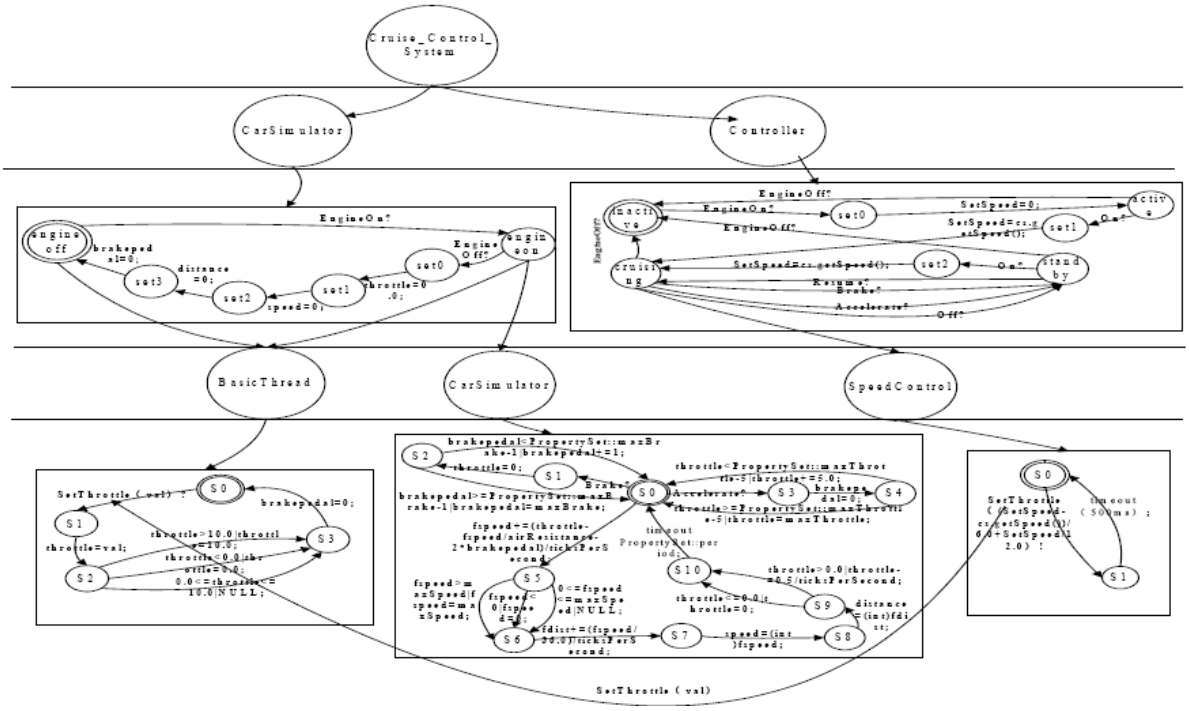

Fig. 1 HTM model of cruise control system.

In Fig. 1, the interaction between the thread SpeedControl and Basicthread is mapped into the link between the action "SetThrottle((SetSpeed-cs.getSpeed())/6.0+SetSpeed/12.0)!" and "SetThrottle(val)?" by Mapping rule 11. The interface automaton which includes "SetThrottle((SetSpeed-cs.getSpeed())/6.0+SetSpeed/12.0)!" corresponds to the behavior specification for the thread SpeedControl. The interface automaton which includes "SetThrottle(val)?" corresponds to the behavior specification for the thread Basicthread.

Step2: the generation of abstract test case set: We save the output and internal action for each abstract test case as expected oracle. So the abstract test case path involves input of test path and oracle of test path. Random algorithm of test case generation is called to 50,000 times. 87 different abstract test cases are derived which cover all states and all transitions.

Step2.1: The tool generates the following part abstract input of test path (That is, the top8) for cruise control system.

- P1:[CarSimulator.EngineOn?] [CarSimulatorThread.Accelerate?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Brake?]
- P2:[CarSimulator.EngineOn?] [CarSimulator.EngineOff?] [CarSimulator.EngineOn?] [CarSimulator.EngineOff?]
- P3: [Controller.EngineOn?] [Controller.On?] [BasicThread.SetThrottle(val)?]
- P4: [Controller.EngineOn?] [Controller.EngineOff?] [Controller.EngineOn?] [Controller.On?] [BasicThread.SetThrottle(val)?] [BasicThread.SetThrottle(val)?]
- P5: [Controller.EngineOn?] [Controller.EngineOff?] [Controller.EngineOn?] [Controller.On?] [Controller.Off?] [Controller.On?] [BasicThread.SetThrottle(val)?] [BasicThread.SetThrottle(val)?]
- P6: [CarSimulator.EngineOn?] [BasicThread.SetThrottle(val)?] [BasicThread.SetThrottle(val)?] [BasicThread.SetThrottle(val)?]
- P7: [CarSimulator.EngineOn?] [CarSimulator.EngineOff?] [CarSimulator.EngineOn?] [CarSimulator.EngineOff?] [CarSimulator.EngineOn?] [CarSimulator.EngineOff?] [CarSimulator.EngineOn?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Accelerate?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Brake?] [CarSimulatorThread.Accelerate?] [CarSimulatorThread.Brake?]
- P8: [Controller.EngineOn?] [Controller.On?] [Controller.Accelerate?] [Controller.Resume?] [Controller.Off?] [Controller.On?] [Controller.EngineOff?] [Controller.EngineOn?] [Controller.On?]

Step2**.2:The tool generates the following oracle of the** above **part test path for cruise control system.**

- E1:[CarSimulatorThread.brakepedal=0;] [CarSimulatorThread.throttle=5.0;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=1;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal+=1;]
- E2: [CarSimulator.throttle=0;] [CarSimulator.speed=0;] [CarSimulator.distance=0;] [CarSimulator.brakepedal=0;] [CarSimulator.throttle=0;] [CarSimulator.speed=0;] [CarSimulator.distance=0;] [CarSimulator.brakepedal=0;]
- E3: [Controller.SetSpeed=0;] [Controller.SetSpeed=cs.getSpeed();] [SpeedControl.timeout(500ms);] [SpeedControl.SetThrottle((SetSpeed-cs.getSpeed())/6.0+SetSpeed/12.0)!] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;]
- E4: [Controller.SetSpeed=0;] [Controller.SetSpeed=0;] [Controller.SetSpeed=cs.getSpeed();] [SpeedControl.timeout(500ms);] [SpeedControl.SetThrottle((SetSpeed-cs.getSpeed())/6.0+SetSpeed/12.0)!] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;]
- E5: [Controller.SetSpeed=0;] [Controller.SetSpeed=0;] [Controller.SetSpeed=cs.getSpeed();] [Controller.SetSpeed=cs.getSpeed();] [SpeedControl.timeout(500ms);] [SpeedControl.SetThrottle((SetSpeed-cs.getSpeed())/6.0+SetSpeed/12.0)!] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;] [BasicThread.throttle=val;]

[BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;]

- E6: [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;] [BasicThread.throttle=val;] [BasicThread.throttle>10.0|throttle=10.0 throttle<0.0|throttle=0.0;] [BasicThread.brakepedal=0;]
- E7: [CarSimulator.throttle=0;] [CarSimulator.speed=0;] [CarSimulator.distance=0;] [CarSimulator.brakepedal=0;] [CarSimulator.throttle=0;][CarSimulator.speed=0;] [CarSimulator.distance=0;] [CarSimulator.brakepedal=0;] [CarSimulator.throttle=0;] [CarSimulator.speed=0;][CarSimulator.distance=0;] [CarSimulator.brakepedal=0;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedalbrakepedal=1;][CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=2;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=3;][CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=4;] [CarSimulatorThread.brakepedal=0;] [CarSimulatorThread.throttle=5.0;][CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=1;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=2;][CarSimulatorThread.brakepedal=0;] [CarSimulatorThread.throttle=5.0;] [CarSimulatorThread.throttle=0;] [CarSimulatorThread.brakepedal=1;]
- E8: [Controller.SetSpeed=0;] [Controller.SetSpeed=cs.getSpeed();] [Controller.SetSpeed=cs.getSpeed();] [Controller.SetSpeed=0;][Controller.SetSpeed=cs.getSpeed();]
-

VI. CONCLUSIONS

Along the formal HTM, and the proposed test case generation, a systematic test generation technique that uses AADL model for the embedded system is proposed and experimented. The technique raises the task of testing AADL implementations using a formal level of automation. This method can not only reduce the work of test cases development and maintenance, move the test process forward and shorten system development time, also guarantee the expected behaviours consistency of system implementation and AADL design through executing test cases which were automatically generated and changed according to AADL system design. In the future work, we can evaluate generated test case by providing concrete values for them to test the large code system, and integrate abstract and concrete test cases into the certification processes for the embedded system.

ACKNOWLEDGEMENTS

This paper is supported by Shaanxi Provincial NSBR Plan in China (No. 2011JQ8008) and Graduation Key Project of Northwestern Polytechnical University.

REFERENCES

[1] SAE, *International. Architecture Analysis and Design Language (AADL*), AS 5506, Nov.2004.

[2] Z. B. Yang, et al. AADL: "An Architecture Design and Analysis Language for Complex Embedded Real-Time Systems," *Journal of Software*, vol.21, no.5, pp.899−915, 2010.

[3] C. Xizhen and Q. Hongbing, "Research on Test Cases Automatic Generation Technique based on AADL Model," *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, pp. 338-342, 2010.

[4] M. Chunyan and D. Yunwei etal, "An Effective Method for Obtaining Architecture of Test Model AADL," *Journal of northwestern polytechnical university*, vol. 28, no. 6, 2010.

[5] SAE AS5506 Annex: Behavior_Specification V2.0 September 20, 2007.

[6] L. D. Alfaro etal. Interface Automata, in the Proceedings of the 9th Annual ACM Symposium on Foundations of Software Engineering (FSE), 2001.

[7] AADL. [Online]. Available: http://www.aadl.info/aadl/currentsite/examplemodel.html