Waterford Institute of Technology

Ireland

Institiúid Teicneolaíochta Phort Láirge

Éire

Validation of Networked Automotive Control Systems Using Global Predicates

---

Zhang Liang （张亮）B.Sc. (Hons)

M.Sc. Thesis

Supervisor: Brendan Jackman B.Sc., M.Tech.

Submitted to the Waterford Institute of Technology

Awards Council, 17 June 2011.

Acknowledgements

I would like to thank sincerely the following people, for all their support and help over the past two years of this project. Without them, this thesis would not be possible.

Firstly, I would like to thank Mr. Brendan Jackman for his encouragement, guidance and patience help throughout this project.

I would like to thank the members in our group for their valuable advice and help.

- Frank Walsh, Group Supervisor, Department of Computing, Math & Physics, Waterford Institute of Technology.
- Rob Shaw, Group member, Department of Computing, Math & Physics, Waterford Institute of Technology.
- Richard Murphy, Group member, Department of Computing, Math & Physics, Waterford Institute of Technology.

I would like to thank Ray Wrynne, who is Sales Director of Vector GB Limited, for offering me the valuable research equipment CANoe.

I would also like to thank Dr. Burkhard Stadlmann and all members of Railway Automation Group of Upper Austria University of Applied Science.

I would lastly and most importantly like to thank my parents who were more of a help than they will ever know.

Abstract

Vehicles consist of many connected networks of electronic control units (ECUs). Automotive application software (e.g. traction control system, climate control, engine management) is distributed across many separate ECUs, making application testing and system integration very difficult. The main difficulty is in constructing a global application state, due to the asynchronous independent operation of each ECU.

The aim of this research is to make the system integration more efficient by creating a global application state based on analysis of application test results. To achieve this goal, a prototype program was developed to construct global states and to analyse these global states. The prototype collects ECU state and network interaction data using the Vector CANoe tool. From these data a lattice of consistent global application states is constructed. The global state lattice can then be used as the basis for analysing ECU signal consistency across ECUs and identifying the potential for erroneous system states to be entered.

Test results from the prototype demonstrate the validity of the theoretical approach despite the disadvantage of the state space explosion associated with large distributed systems.

Declaration

I, Liang Zhang, declare that this thesis is submitted by me in partial fulfilment of the requirement for the degree M.Sc., is entirely my own work except where otherwise accredited. It has not at any time either whole or in part been submitted for any other educational award.

Signature: _____

Liang Zhang,

17 June 2011.

Table of contents

Table of figures

Table of tables

Section One: Introduction

Chapter 1 Introduction

1.1 Automotive integration problem

Modern vehicles contain many electronic control systems to enhance fuel efficiency, engine performance, vehicle chassis control and passenger comfort, as well as reducing emissions. These control systems are organised as multiple interconnected networks of distributed software components running on many Electronic Control Units (ECUs). In order to efficiently develop the application software and aid the mobility of software components, a kind of middleware AUTOSAR is developed. It is used as a standard runtime platform for the automotive software components. AUTOSAR is very similar to the common middleware CORBA (Common Object Request Broker Architecture) or JRE (Java Runtime Environment) except that the services provided are highly specialised and optimised for the automotive environment.

Although these OS and middleware offers huge help for the development of the automotive software, it still does not sort the problem of the integration of the ECUs. These standards are very handy for a single ECU, but eventually all these ECUs will be assembled into a vehicle, communicating through the network. They cooperate with each other to achieve the user requirements. There are a few types of integration test for the automotive software: MIL (Model In the Loop), SIL (Software In the Loop), and HIL (Hardware In the Loop). The most critical one is HIL which tests and validates software on the real hardware ECUs. All these ECUs are distributed, so the natural difficulty for testing the distributed system is present. The difficulty is caused by the property of the distributed system; there is no global clock and shared memory for the distributed system, so

it is very difficult to record and validate the concurrent states of multiple ECUs. Examples of ECU states might be “sending value of current speed of the vehicle” “waiting to receive temperature of the engine”, “processing power steering angle” and so on.

Figure 1-1 shows a network structure of a modern vehicle (Gabriel Leen and Donal Heffernan 2002).

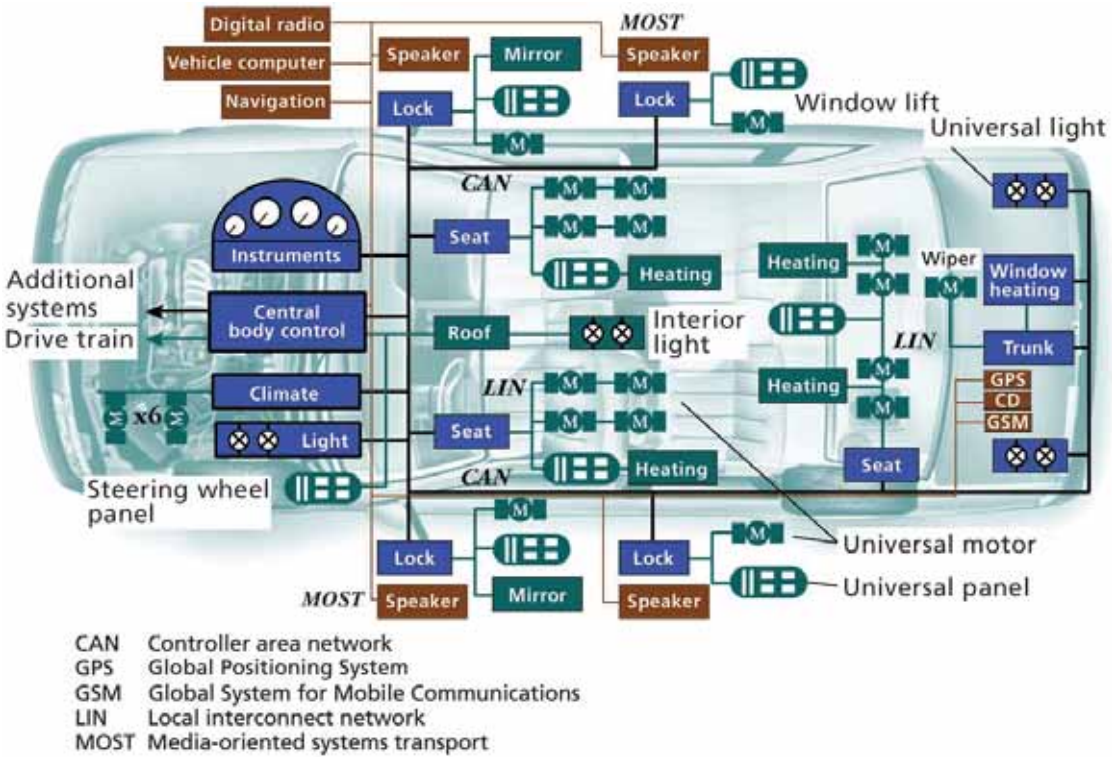


Figure 1-1 automotive network application

1.2 Research questions

The main goal of this research is to investigate the theoretical methods for constructing the global state of a system, made up of networked ECUs on a CAN bus.

The key questions of this research are as follows:

- How can events occurring on separate ECUs be chronologically ordered?
- How can a snapshot of the global application state and application execution traces be constructed based on test case execution cycles?
- How to deal with the large number of potential global states of execution?

1.3 Document layout

Chapter 1: Introduction

This chapter introduces the objective of the research and discovers the problem. The research questions are addressed.

Chapter 2: Automotive Application Development

This chapter describes the operating system and network protocol used in the automotive industry.

Chapter 3: Automotive Software Testing

This chapter introduces the software testing methodologies and how the automotive industry does software testing.

Chapter 4: Logical Time

This chapter introduces how to order the events in the distributed system.

Chapter 5: Global State and Snapshot

This chapter describes the global state and algorithms to do a snapshot to record the global state for the distributed system.

Chapter 6: Global State Evaluation

This chapter describes the algorithms to continually record global states of the system and global predicate evaluation algorithms.

Chapter 7: Methodology

This chapter describes the method is applied to solve the automotive integration problems.

Chapter 8: Prototype Development

This chapter introduces the development of the prototype program, and the structure and the function activities of the prototype.

Chapter 9: Prototype Testing

This chapter list the test cases to verify and validate the prototype software.

Chapter 10: Research Conclusion

This chapter summarises this research, answers the research questions, and gives the potential for the further development and research.

References

Gabriel Leen & Donal Heffernan. Expanding Automotive Electronic Systems. 88-93. 2002. IEEE.

Section Two: Literature Review

Chapter 2 Automotive Application Development

2.1 Introduction

This chapter introduces automotive application development. As desktop computer, the application software task and hardware resources are managed by the operating system; for the automotive ECU such operating system is also needed. The most popular automotive operating system used nowadays is OSEK/VDX OS (AUTOSAR GbR 2008c). OSEK/VDX (Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug/ Vehicle Distributed eXecutive) is set of standards for distributed automotive systems. It was developed by German and French automotive manufactures -(Joseph Lemieux 2001). The OSEK/VDX includes four main standards: operating system (OS), communication (COM), network management (NM), OSEK implementation language (OIL). It also includes three additional standards: OSEK/VDX real-time interface (ORTI), OSEK/VDX time-triggered operating system (OSEK-Time), and OSEK/VDX fault tolerant communication specification. Also there is another new standard getting popular in the automotive industry. It is called AUTOSAR (AUTomotive Open System ARchitecture). AUTOSAR is an open standardized software architecture for the automotive industry (Simon Fuerst and BMW 2008). It separates the system into different layers: application layer, AUTOSAR Runtime Environment (RTE), service layer, ECU abstraction layer, Microcontroller abstraction layer, and complex devices. RTE isolates the application layer from the other layers (AUTOSAR GbR 2008a), the developer does not need to deal with the hardware drivers. It also enhances code mobility and compatibility and reduces the development complexity. This chapter will talk

about the OSEK/VDX OS, AUTOSAR OS, OSEK/VDX COM and AUTOSAR COM. They are relevant to this research.

There are many different ECU networks e.g. CAN, FlexRay, LIN, and MOST etc. This research only focuses on the CAN protocol network, which will be described in this chapter. For reading and writing the ECU memory, the CAN Calibration Protocol (CCP) is also introduced in this chapter.

Figure 2-1 shows a typical distributed automotive application

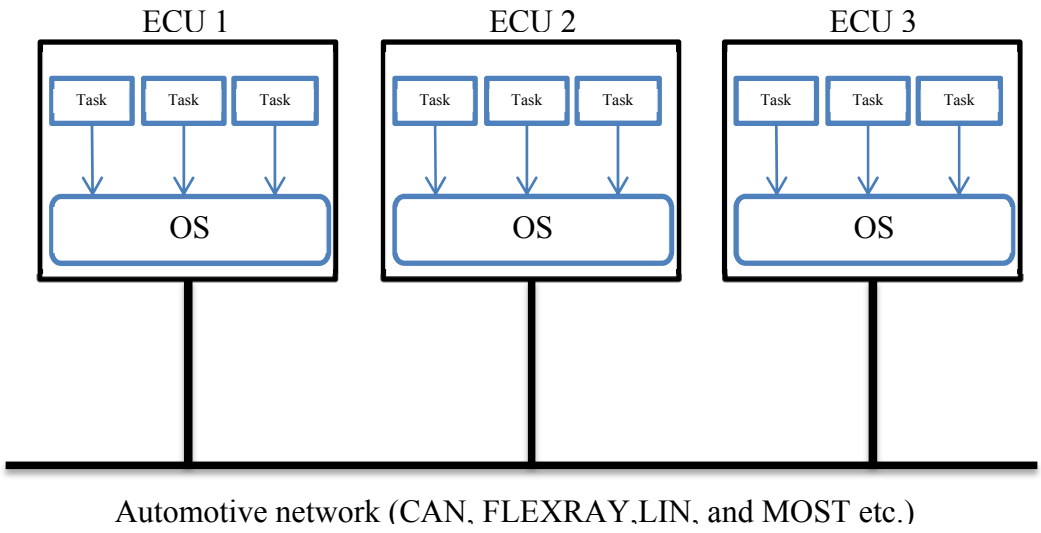


Figure 2-1 distributed automotive application

2.2 ECU software Functions

The embedded application is implemented as a set of event-driven functions. The event-driven functions are triggered by an event-e.g. sensor, timer expired, and message received etc.. These events may be detected by either interrupt or polling.

Automotive application program functions are currently implemented as task on OSEK based operating systems (such as Ford's FNOS or Vector CANbedded) or

as AUTOSAR components. This section will introduce the OSEK OS and AUTORSAR OS.

2.2.1 OSEK OS

2.2.1.1 Overview of OSEK operating system

The OSEK operating system provides a pool of different services and processing mechanisms (OSEK 2005). According to the configuration required by the user, the OSEK operating system is built at system generation time.

The OSEK/VDX OS manages the application programs which are independent of each other for the processor. It schedules the work of the processor by assigning the application to different processing levels.

The essential concept in the OSEK/VDX OS is the task. There are two types of tasks: basic task and extended task. The activation of a task depends on the priority of the task.

OSEK OS offers four conformance classes. Depending on the requirement of the application software and system resources (e.g. processor, memory), the conformance class describes the available features of the operating system.

2.2.1.2 Processing levels

OSEK defines three processing levels:

1. Interrupt level
2. Logical level for schedules
3. Task level

The interrupt level processes have the highest priority over other processes. And the task level has the lowest priority. The interrupt process level includes one or

more interrupt priority levels. Interrupt service routines have a statically assigned interrupt priority level. Assignment of interrupt service routines to interrupt priority levels is dependent on implementation and hardware architecture. For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities. The task's priority is assigned by the user (the task priorities are introduced in section 2.2.1.6).

2.2.1.3 Basic task

A basic task runs to completion unless preempted by a higher priority task or an interrupt (if enabled)(Joseph Lemieux 2001). It has three states as shown in Figure 2-2.

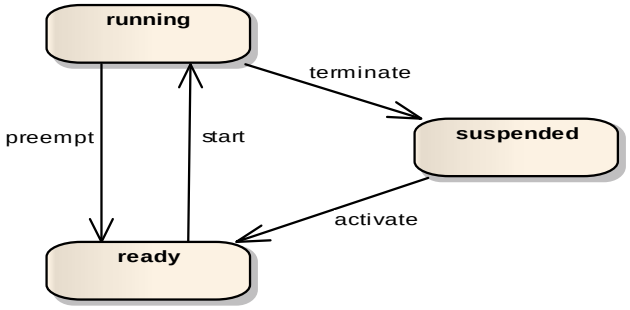


Figure 2-2 basic task state transition

The states are explained in Table 2-1. The transitions are explained in Table 2-2.

Running	Allocating the processor to process the task. The instruction of the task is executed. Only one task can be in this state at any time. The other states can be adopted simultaneously by several tasks
Ready	Waiting for allocating the processor to the task.
Suspended	In the suspended state, the task is passive and can be activated.

Table 2-1 basic task states explanation

Transition	Former state	New state	Description
activate	suspended	ready	A new task is set into the ready state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	ready	running	A ready task selected by the scheduler is executed.
preempt	running	ready	The scheduler decides to start another task. The running task is put into the ready state.
terminate	running	suspended	The running task causes its transition into the suspended state by a system service.

Table 2-2 basic task transitions explanation (OSEK 2005)

2.2.1.4 Extended task

The extended task is very similar to basic task. The only different is that the extended task has one more state called waiting state. The state diagram for extended task is illustrated in Figure 2-3. Waiting state is used by the task that cannot continue execution until an event triggers it. The other states are as same as basic task states. The transitions are described in Table 2-3.

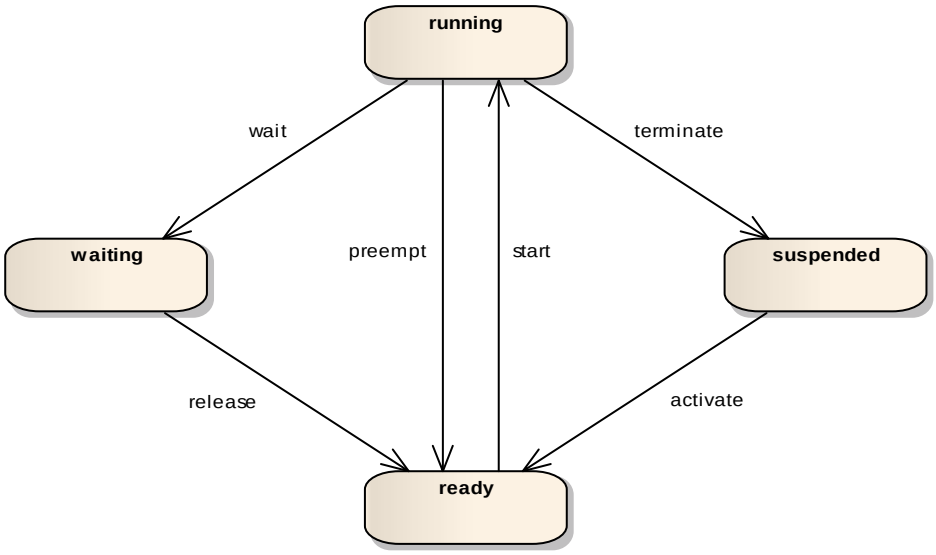


Figure 2-3 extended task state transition

Transition	Former state	New state	Description
activate	suspended	ready	A new task is set into the ready state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	ready	running	A ready task selected by the scheduler is executed.
wait	running	waiting	The transition into the waiting state is caused by a system service. To be able to continue operation, the waiting task requires an event.
release	waiting	ready	At least one event has occurred which a task has waited for.
preempt	running	ready	The scheduler decides to start another task. The running task is put into the ready state.
terminate	running	suspended	The running task causes its transition into the suspended state by a system service.

Table 2-3 extended task state transitions explanation(OSEK 2005)

2.2.1.5 Comparison of basic task and extended task

The basic task does not have a waiting state. The synchronization points are formed at the task start and end. If the application needs internal synchronization points, then more than one basic task is required. The advantage of basic tasks is that they do not use too much RAM. This because basic task does not have the waiting state; in a waiting state, the task is loaded into the RAM to wait for an event to active it. The advantage of extended task is that even though the synchronization is requested, one task can deal with a coherent job. When the extended task needs the data (new data or updated data) to continue execution, it will be in the waiting state, until the requested data arrives (Joseph Lemieux 2001).

2.2.1.6 Task priority

Every task in OSEK/VDX OS has a priority. The priority is statically assigned to the task, it cannot be changed dynamically. There is one situation that the priority can be changed by the OS. It happens when the priority ceiling protocol is active: the priority of a task is elevated to the priority ceiling value calculated statically (Joseph Lemieux 2001).

The value 0 is the lowest priority of a task. The larger number has the higher priority (OSEK 2005). The same priority tasks can be grouped together. They are stored in a FIFO queue.

2.2.1.7 Scheduling policies

A task, whether basic or extended, can be set as either full preemption or non-preemption. The non-preemption task runs until it terminates or, in the case of extended task, until it transitions to a waiting state. When a preemption task is running, it can be preempted by a task with higher priority task. (Joseph Lemieux 2001)

Depending on the attribute of preemption of the task, OSEK/VDX OS schedules the tasks. The scheduling policy consists of full preemptive scheduling, non-preemptive scheduling, and mixed preemptive scheduling. They will be introduced in the following sub-sections.

2.2.1.7.1 Full preemption scheduling

Full preemptive scheduling means that a running task is put into a ready state by a higher priority task. Figure 2-4 demonstrates full preemptive scheduling. Task1 has higher priority than Task2. When Task2 is running, Task1 starts. Because the task1 has higher priority and Task2 is preemptive, Task2 is preempted, task 1

cannot be delayed.

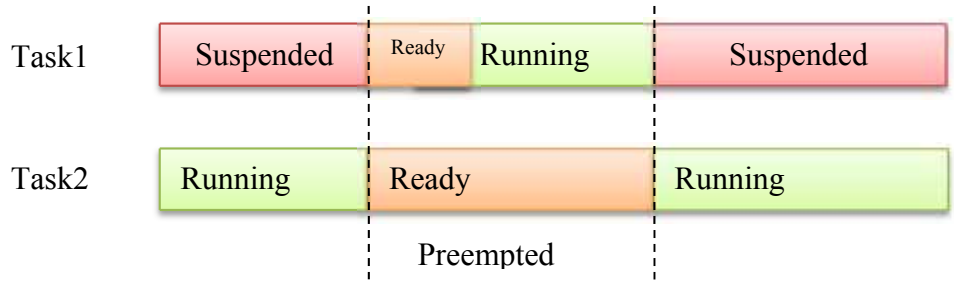


Figure 2-4 Full preemptive scheduling

2.2.1.7.2 Non-preemptive scheduling

Non-preemptive scheduling occurs when the current running task cannot be preempted by another task even a task with higher priority. So the higher priority task can be delayed by the non-preemptive and low priority task. Figure 2-5 demonstrates a non-preemptive scheduling. Task1 has higher priority than Task2. When Task2 is running, Task2 is started, it is only stay in the ready state, until the Task2 is terminated, then Task1 moves to running state. The delay time of Task1 is the time that it is in the ready state.

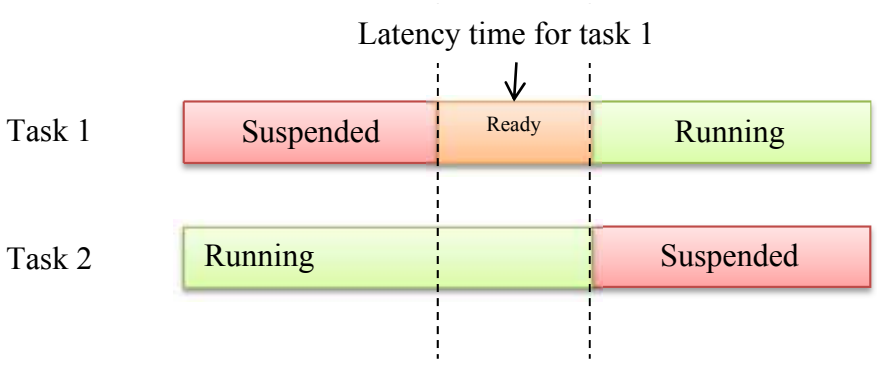


Figure 2-5 non preemptive scheduling

2.2.1.7.3 Mixed preemptive scheduling

The mixed preemptive scheduling system mixes the preemptable and the non-preemptable tasks. The scheduling policy depends on the preemptive attribute of individual task. If the running task is preemptive, then the full preemptive scheduling will be applied. If the running task is non preemptive, then the non preemptive scheduling will be applied.

2.2.1.7.4 Conformance classes

Depending on the requirement of the application software for the system and the abilities of the system (e.g. processor, memory), the operating system features can be configured. These features used to describe the operating system are called conformance classes (CC).

There are 4 conformance classes defined (OSEK 2005):

1. BCC1 (only basic tasks, limited to one activation request per task and one task per priority, while all tasks have different priorities)
2. BCC2 (like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed)
3. ECC1 (like BCC1, plus extended tasks)
4. ECC2 (like ECC1, plus more than one task per priority possible and multiple requesting of task activation allowed for basic tasks)

The determination of conformance class is illustrated in Table 2-4.

Attribute	BCC1	BCC2	ECC1	ECC2
Number of basic task activations	1	≥1	1	≥1
Number of tasks per priority	1	≥1	1	≥1
Basic tasks	Yes	Yes	Yes	Yes
Extended tasks	No	No	Yes	Yes

Table 2-4 conformance class determination

The conformance classes are upwardly compatible as shown in Figure 2-6

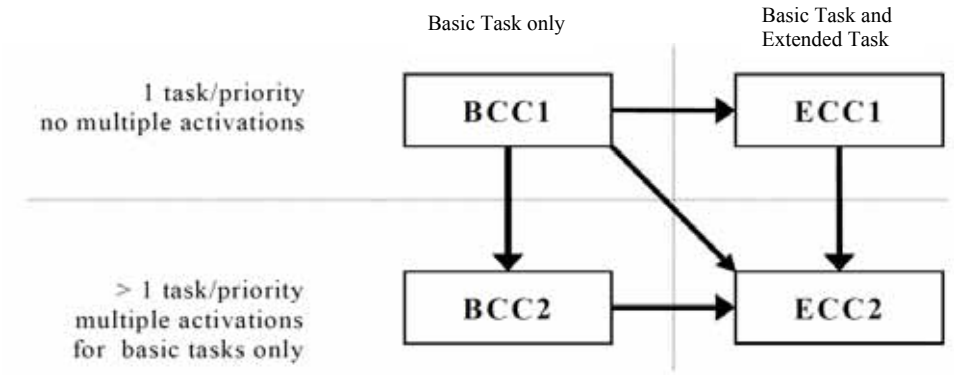


Figure 2-6 upward compatibility for conformance classes

Figure 2-6 shows any task developed for a BCCx level conformance class can be used in an ECCx-level conformance class and any task developed for a xCC1 level conformance class can be used in an xCC2 level conformance class.

2.2.2 AUTOSAR

2.2.2.1 Overview of AUTORSAR

The purpose of AUTOSAR is to standardize the software architecture of ECUs. It makes the software independent from the hardware. The horizontal layers means the development can be processed simultaneously and thereby reduce the development time and costs. The software will be more reusable for OEM

(Original Equipment Manufacturer) as well as for suppliers. It enhances quality and efficiency.

Figure 2-7 shows the comparison of AUTOSAR architecture and previous architectures (Stefan Bunzel 2010).

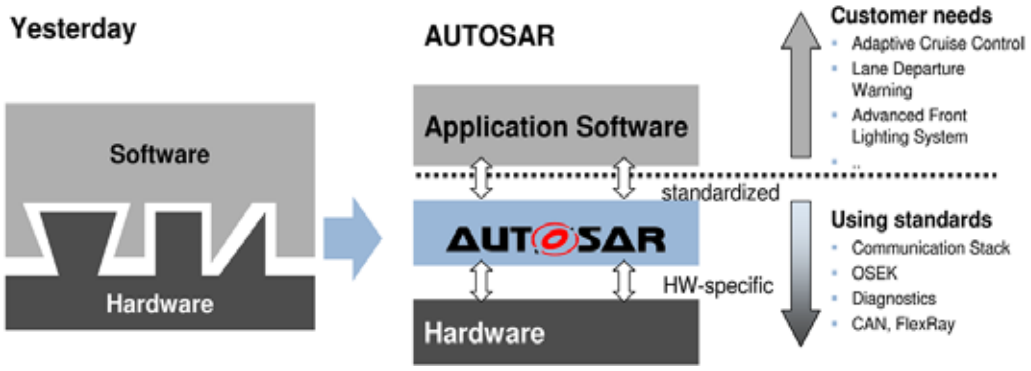


Figure 2-7 comparison between AUTOSAR architecture and older architectures

AUTOSAR defines the software as different layers. There are 5 layers in the AUTOSAR architecture as show in Figure 2-8(AUTOSAR GbR 2008a). The application software makes up the components in the application layer.

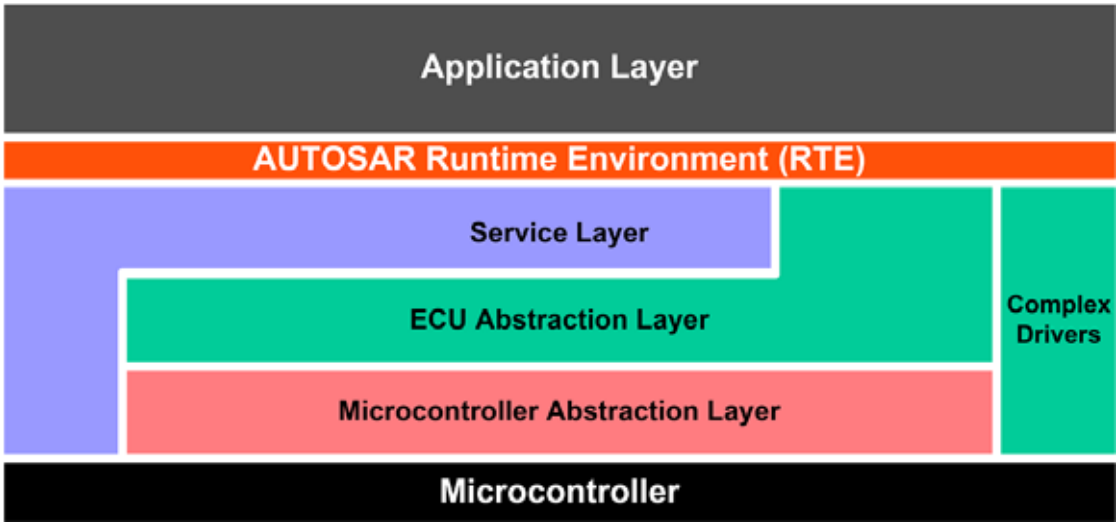


Figure 2-8 AUTOSAR layers

The Microcontroller (MCU) Abstraction Layer is the lowest software layer. It contains the internal drivers, which are software modules with direct access to the MCU internal peripherals and memory mapped MCU external devices.

The ECU Abstraction Layer interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an API for access to peripherals and devices regardless of their location (MCU internal/external) and their connection to MCU (port pins, type of interface).

The Service Layer provides basic services for each AUTOSAR application. An AUTOSAR application can access these services through standardized AUTOSAR interfaces(Robert Warschofsky 2011).

The ECU abstraction layer and the service layer, together are called Basic Software layer (BS). So the AUTOSAR also can be described as a 4 layered system as shown in Figure 2-9 (Simon Fuerst & BMW 2008).

The RTE Layer provides a running environment that makes the application program independent from the ECU. When a AUTOSAR application program is completed, it can be run on different ECUs in which AUTOSAR is installed without change in code(AUTOSAR GbR 2010).

The Application Layer holds the application task as a set of components. The components can communicate with each other through the AUTOSAR interface as shown in Figure 2-9. The components also can be in the different ECUs. All the layers that are lower than application layer will deal with the network, hardware drivers and the system services etc. Therefore, the developer can only focus on the application software development. AUTOSAR using standard

interface connects the application to the RTE layer. It makes the function transferable and the code reusable.

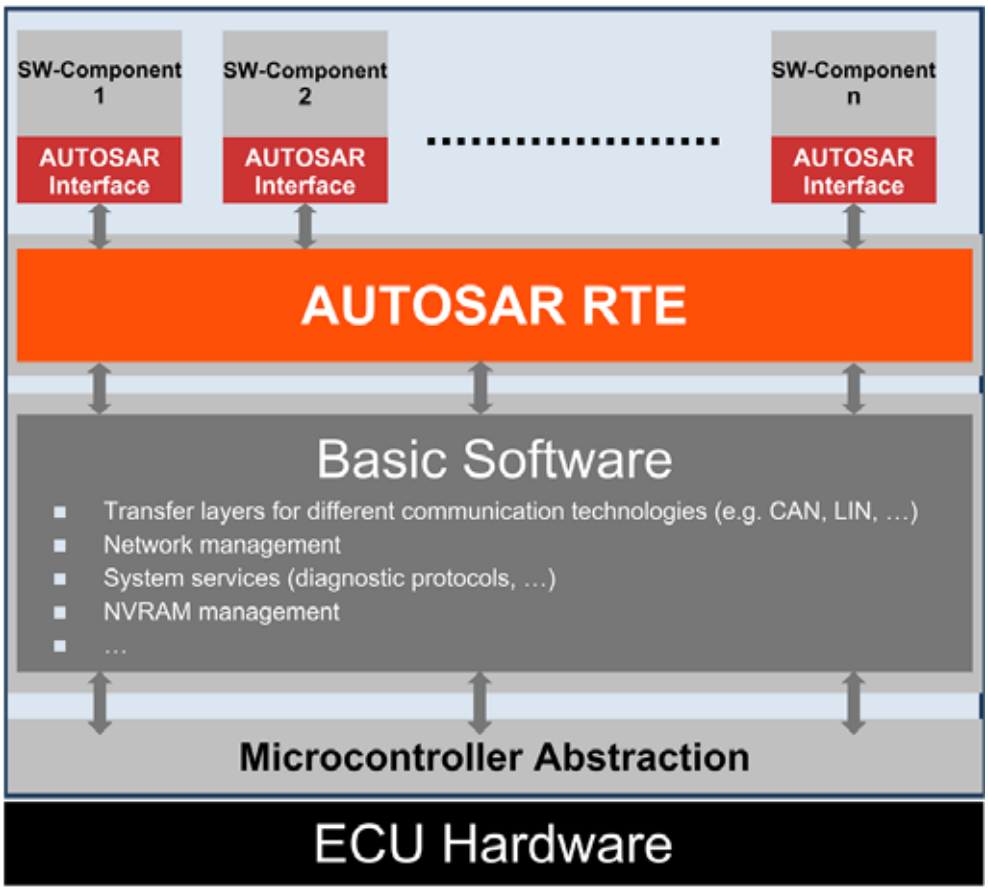


Figure 2-9 AUTOSAR basic software

2.2.2.2 *Software component*

In the AUTOSAR system, the application is divided into functions. Each function is encapsulated in the AUTOSAR software component. Because the application is constructed by the components, it makes the component more reusable, and a different application may use the same component/s. The components interacts with each other through the Virtual Functional Bus (VFB) to implement the application. In the VFB model, software components interact on interfaces

between ports. The port/interface model is called an AUTOSAR interface. The view of VFB is illustrated in Figure 2-10 (Darren Buttle 2005).

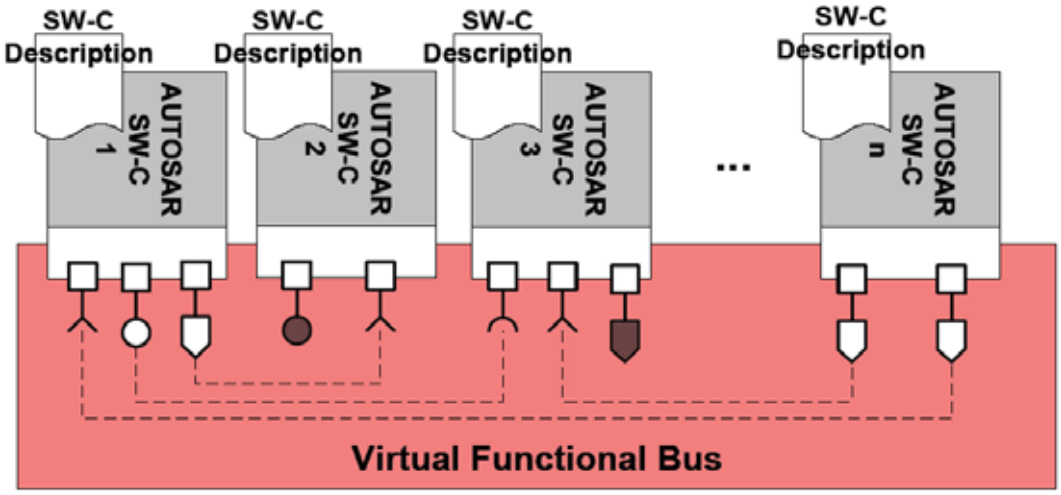


Figure 2-10 VFB view

There are two types of AUTOSAR component: atomic software components and Sensor/Actuator Software Component. Atomic software components implements a piece of software that can be mapped to an AUTOSAR ECU. Hardware sensor/actuator is coupled to sensor/actuator software component.

Components have two types of interface (ports): provided interfaces and required interfaces. The provided interface gives the function or data through P-port. The required interface needs the function or data through R-port. The component

interfaces can be illustrated in Figure 2-11 (Darren Buttle 2005).

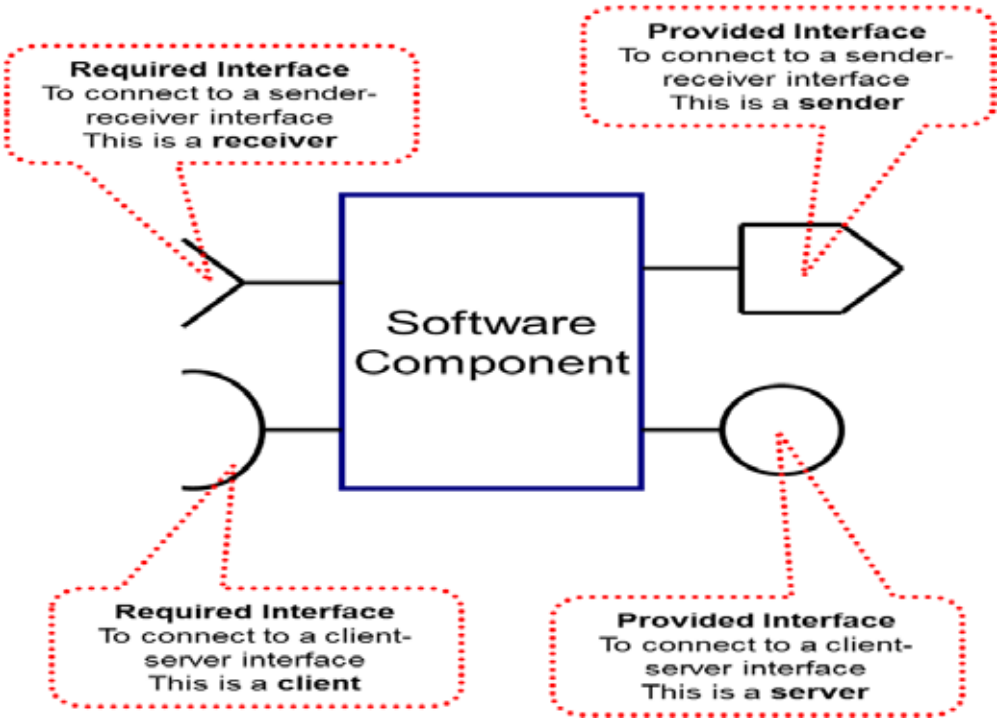


Figure 2-11 AUTOSAR software component with interfaces

All communication in AUTOSAR modelled between ports are sender-receiver (signal passing) and client-server (function calling) as show in Figure 2-11.

2.2.2.3 Sender-receiver

Sender-receiver communication: the data is transmitted by one component and received by one or more components. A component can have multiple sender-receiver interfaces. Each sender-receiver interface can have multiple data elements. Each data element can be sent or received independently. The data can be simple types (integer, float) or complex (array, record). Figure 2-12 shows the sender side of a sender-receiver interface that includes three data simple elements. Components can use “1:1”, “n:1” and “1:n” communication. (LiveDevices Ltd. 2004) , p30-31)

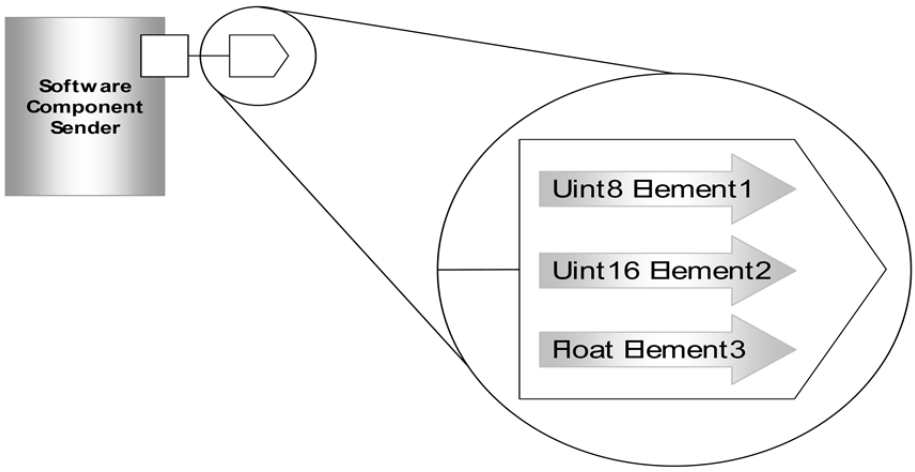


Figure 2-12 Sender-Receiver Interface Data Elements (Sender Side)

RTE supports multiple receive modes for the receiver software component to handle the received data. The four possible receive node are: Implicit data read access, Explicit data read access, wake up of wait point, and activation of runnable entity. (AUTOSAR GbR 2010), p108-109)

- Implicit data read access: when the receiver's runnable executes it shall have access to a "copy" of the data that remains unchanged during the execution of the runnable.
- Explicit data read access: the RTE generator creates a non-blocking API (Application Programming Interface) call to enable a receiver to poll (and read) data. This receive mode is an "explicit" mode since an explicit API call is invoked by the receiver.
- Wake up of wait point: the RTE generator creates a blocking API call that the receiver invokes to read data. Runnable awoken when the data received.
- Activation of runnable entity: the receiving runnable entity is invoked automatically by the RTE whenever new data is available.

These receive modes also can be applied on client-server communication, if clients call server asynchronously.

2.2.2.4 Client-server

Client-server communication: a client component invokes function of the sever component. A component can have multiple client-server interfaces. Each client interface can have multiple operations. Each operation can be invoked separately. Figure 2-13 shows the server side of a client-server interface that serves elementary sorting algorithms to the client. Components support “1:1”, “n:1” communication. Clients cannot have multiple servers. (LiveDevices Ltd. 2004), p32-33)

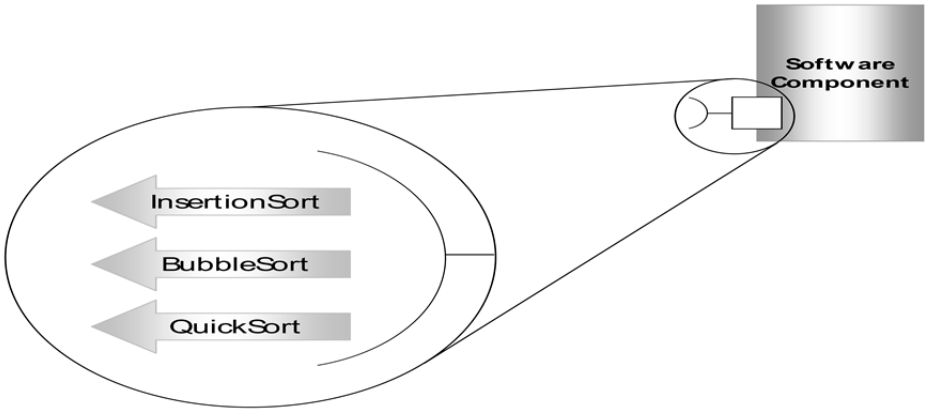


Figure 2-13 Client-Server Interface Operation (Server Side)

The client-server interfaces can control how the server buffers client requests for the operations. In the no buffering server, the server will reject and send back an error notification to the requesting client while it is processing the early request. In the server with buffering, the server will queue the client requests. The size of the queue is predefined at the configure time. If the queue is full, the new request will be discarded without any error reply to the client.

2.2.2.5 Internal communication

Sender-receiver and client-server communication through AUTOSAR ports are the model for communication between software components. For an individual component, it can contain one or more runnable entities ("runnables"). A runnable and the task is the same thing. These runnables will collaborate to each other to achieve the function of the component. A runnable is an entry point of the function as well as the subroutine of the program. Runnables has two categories as following (AUTOSAR GbR 2006a, p16-17):

1. Category 1: runnable entities do not have wait point (wait state) and have to terminate in finite time. It can be divided to two part:
    a. The runnable entity is only allowed to use implicit reading and writing. A category 1a runnable entity cannot block and cannot use explicit read/write.
    b. The runnable entity may use explicit reading/writing including blocking behavior.

    Category 1 is very similar to the basic task model in OSEK OS.

2. Category 2: It always has at least one wait point or they invoke a server and wait for the response to return. It is very similar to the extended task model in OSEK OS.

Depending on the way to activate the runnable, a runnable can be either timing or event triggered (e.g. real time clock alarm expires) or communication triggered (e.g. a signal received). Figure 2-14 shows a component and runnables (Darren Buttle 2005).

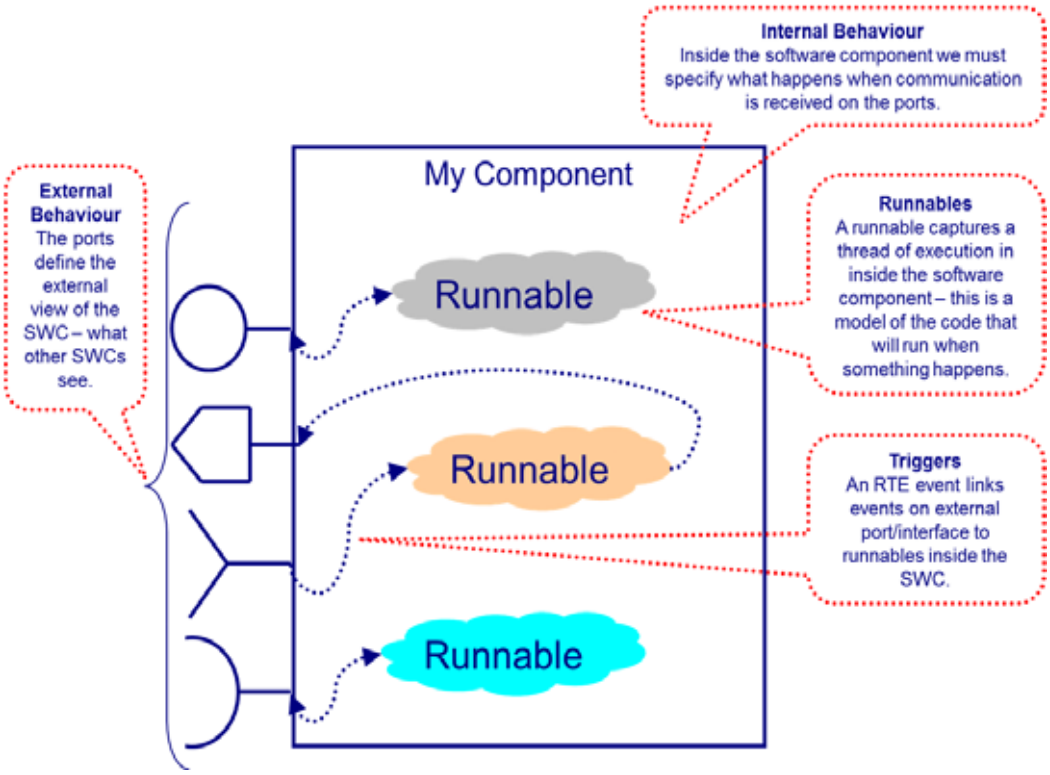


Figure 2-14 software component and runnables

2.2.2.6 Composition

The AUTOSAR composition aggregates existing software components to form another function. Therefore, a composition is also a kind of component. The composition may be aggregated in even further compositions (AUTOSAR GbR 2006b, p27). Such recursive relationship is illustrated in Figure 2-15. The compositions make code more reusable and enhance the mobility of the code.

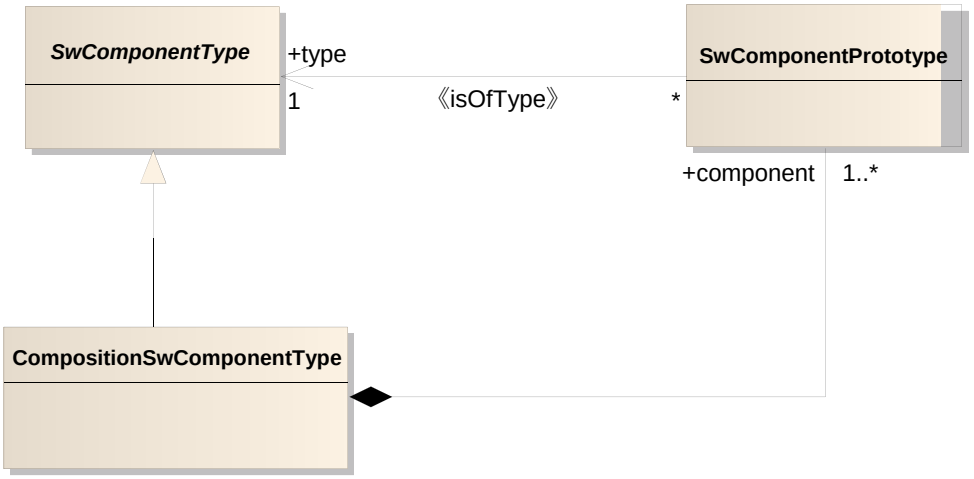


Figure 2-15 the recursive relation of software components and compositions

The compositions offer logical software architecture. Because the VFB maps the software component to the hardware, the composition can be designed independently. During the development, the target hardware and network topology etc. do not need to be referenced. This is the big advantage compared to ECU-driven development. Without concerning the hardware, the logical functionalities can be constructed. The functions can be integrated together to perform a new function. The logical functions integration also can be done at very early stage.

2.3 ECU networks

The previous section describes a single ECU system. However, for the automotive control system, multiple ECUs are used. They collaborate with each other to achieve a common goal. There are some different protocols for the ECU network, e.g. CAN, FlexRay, LIN,etc. Nowadays, the most popular protocol used is CAN

(Controller Area Network) based network (Nicolas Navet and Françoise Simonot-Lion 2009, p X). The main difference between CAN and FlexRay is that CAN is an event-triggered network and FlexRay is a time-triggered network.

The big difficulty of debugging distributed systems is that there is no global time in the distributed system. However, in the FlexRay network system, there is a distributed clock synchronisation mechanism. Each node synchronizes itself to the global time of the cluster by measuring the timing of transmitted sync frames sent by sync nodes (Richard Zurawski 2009, p167-169). There is a global clock in the FlexRay networked system. Each node is synchronized by the global clock; thereby at any global time point, the global state can be constructed by the local state of each node at an agreed global time. The global states can be ordered by the global time. Even with the global time, the distributed system can be debugged as the single CPU system. The break point can be set, the whole system will stop at the same time (global time). Therefore, it is much easier to debug the distributed automotive system with FlexRay.

For a CAN network based distributed system, there is no mechanism to synchronize all nodes on the network. The transmission of the message depends on the priority of the message. It is very difficult to debug a CAN based distributed system. This research will focus on CAN networked automotive systems. This section introduces the CAN protocol.

2.3.1 CAN network features

CAN is a bus structure network. Each ECU on the CAN bus has the same priority to send messages, so it is a multi-master network. A CAN message has a unique identifier that specifies its content and transmission priority. The messages are

broadcast and multicast on the CAN bus. The CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) principle is used by CAN bus, the CAN network offers Non-destructive bus arbitration scheme. CAN bus has comprehensive error detection and confinement (Brendan Jackman 2004a, p1).

2.3.2 CAN bus structure

The CAN network structure is illustrated in Figure 2-16. Each ECU node connects to a two wire network. The two wires are twisted together to reduce the electromagnetic interference. At two ends of the bus are 120 Ω resistors to remove signal reflections. Since the CAN bus is a digital bus, it is always at logical 0 or logical 1. Zero is known as the dominant level, one is known as the recessive level.

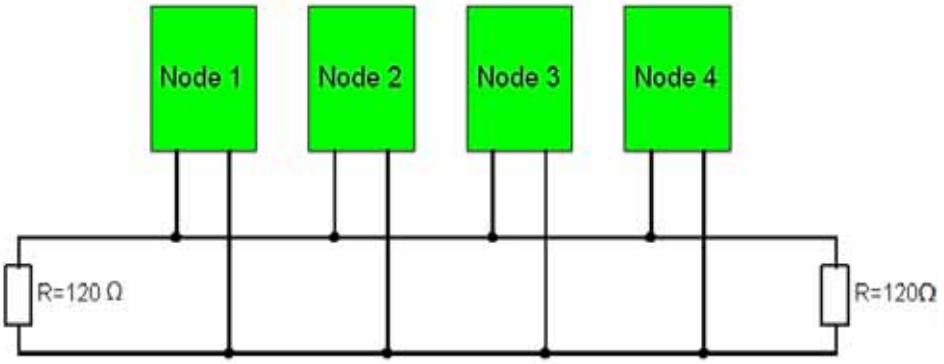


Figure 2-16 CAN network structure

2.3.3 Dominant & recessive bits

If more than one node on the CAN bus which to transmit a message at the same time, the message with the dominant (zero) bit will automatically overwrite the message with the recessive (one) bit. When the CAN bus is idle, it is at a recessive

level. Dominant and recessive both play a role in prioritizing messages during bus arbitration.

2.3.4 CAN Frames

CAN networks contain following types of frames (BOSCH 1991):

- **A Data Frame** carries data from a transmitter to the receivers.
- **A Remote Frame** is transmitted by a bus unit to request the transmission of the data frame with the same identifier.
- **An Error Frame** is transmitted by any unit on detecting a bus error.
- **An Overload Frame** is used to provide for an extra delay between the preceding and the succeeding data or remote frames.

There are two different types of data frame: standard and extended data frame. The only difference between them is the length of their arbitration field, which will be described in this section. Because the data frame and remote frame are very similar, they will be described in the same section. The overload frame is rarely used nowadays.

2.3.4.1 Data frame and remote frame

A CAN message data frame is illustrated in Figure 2-17.

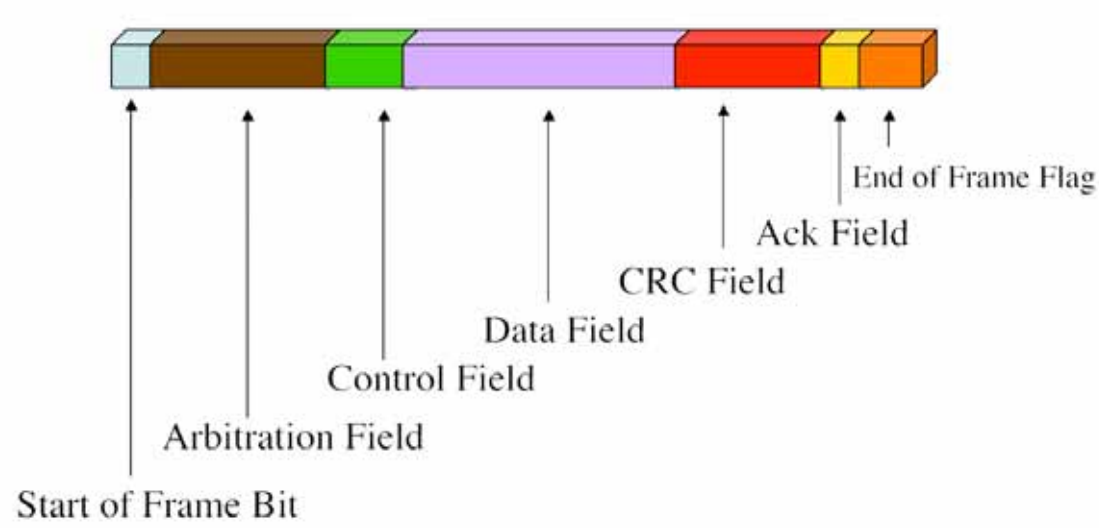


Figure 2-17 CAN message data frame (Brendan Jackman 2004b)

Start of Frame: it contains single dominant bit, telling the CAN bus a message is going to be transmitted.

Arbitration Field: the difference between the standard and extended data frame is the length of their arbitration field. For the standard data frame, its arbitration field can be separated into two parts: message identifier (Id) (11 bits) and remote transmit request (RTR) (1 bit), and is illustrated in Figure 2-18. If the RTR is dominant, then the message is a data message, otherwise the message is remote message. The arbitration field of the extended data frame is illustrated in Figure 2-19. The message ID is separated by two fixed recessive bits which are the Substitute Remote Request bit (SRR) and IDentifier Extension bit (IDE). The most significant 11 bits of the message ID are transmitted first. The other 18 bits of the message ID follow the IDE bit. The last bit of the arbitration field is the RTR bit.

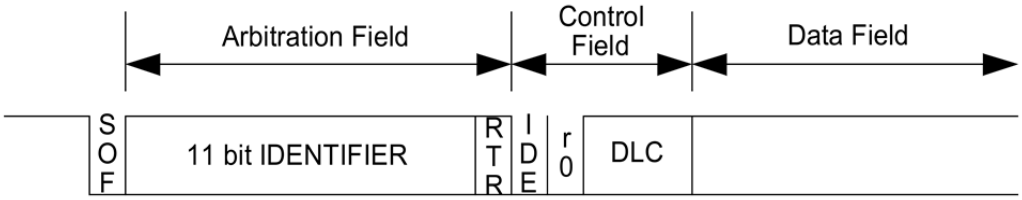


Figure 2-18 standard data frame

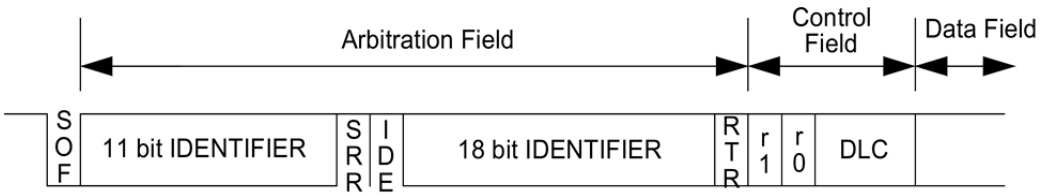


Figure 2-19 Extended data frame

Control field: it defines the type (standard or extended) and the length of the transmitted message. If the message is standard, then the control field has the IDE bit, the reserved bit r0, and the Data Length Code (DLC) (4 bits) as shown in Figure 2-18. If the message is extended, then the control field has two reserved bits (r1 and r0) and DLC, as shown in Figure 2-19.

Data Field: it contains the data that needs be transferred. It can contain from 0 to 8 bytes of data.

CRC (Cyclical Redundancy Check) Field: it holds a 15-bit number that is calculated based on the data of start of frame, arbitration field, control field, and data field. After this 15-bit number, a recessive bit marks the end of the CRC field.

Acknowledgement (ACK) field: it contains two bits (ACK slot and ACK delimiter). It checks if any node received the message. If a node received the message and the CRC correctly, the node overwrites the ACK slot that the transmitter sets recessive with a dominant bit. If the transmitting node does not

see the ACK slot marked as dominant, then it will transmit the message again. The ACK delimiter is always recessive.

Remote Frame: it is very similar to data frame, the only different is their RTR bit of the arbitration field. The RTR bit of remote frame is recessive.

End of Frame Flag: it consists of 7 recessive bits that marks the end of the CAN message.

2.3.4.2 Error frame

An error frame is illustrated in Figure 2-20. The Error Flag consists of either 6 dominant bits (active error flag) or 6 recessive bits (passive error flag). The Error Delimiter consists of 8 recessive bits. It is used to signal the presence of errors on the bus.

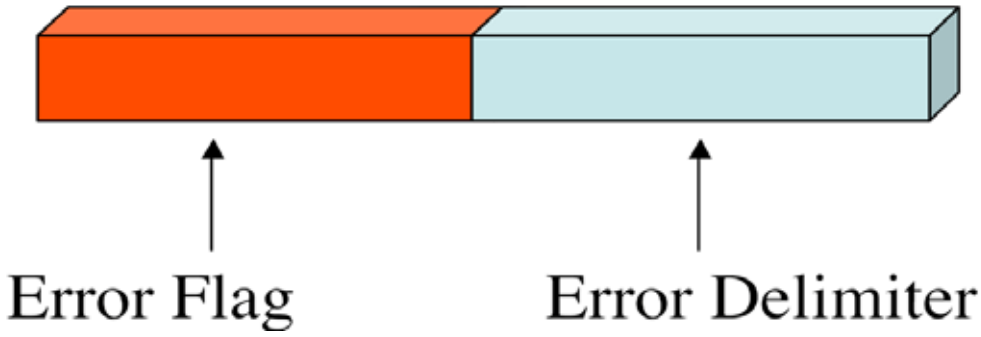


Figure 2-20 error frame

2.3.4.3 *Overload frame*

An overload frame consists of two bit fields, overload flag and overload delimiter. Its structure is same to error frame. An overload frame is illustrated in Figure 2-21. The overload flag includes 6 dominant bits. The overload delimiter consists of 8 recessive bits. It is used to delay the message transmission.

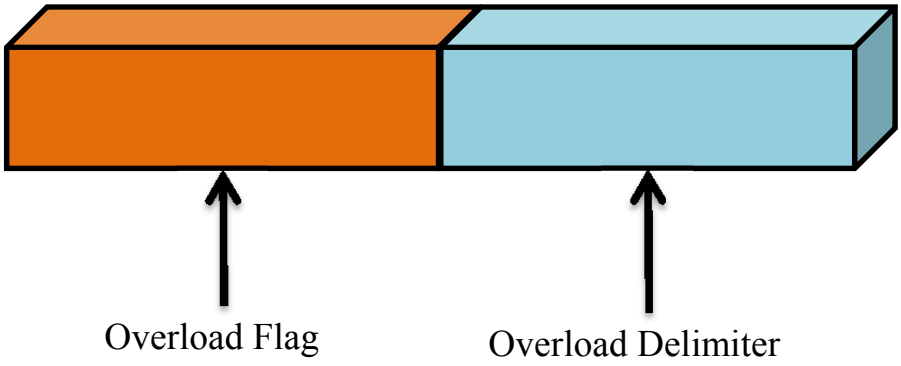


Figure 2-21 overload frame

2.3.5 CAN arbitration

The CAN arbitration happens when more than one node sends a message at the same time. The transmitting node will compare its arbitration field of the message to other nodes bit by bit. The lowest CAN ID wins the arbitration. If the CAN ID is same, the standard message wins the extended message and the data message wins the remote message (Richard Zurawski 2009, p137). However, this typically does not happen.

2.3.6 Error handling

There are two levels of error for the CAN bus: message level error and node level error.

Message level error is caused by the inconsistent message formatting. e.g. fail of CRC checking, the format of the message layout is corrupted. If the message level error happens, an error frame message is transmitted.

Depending on the error counter's level, the node can be in one of the following states: error active, error passive, and bus off. Each node has a Transmit Error Counter (TEC) and a Receive Error Counter (REC) that determines the node state. TEC increases when a transmitting node detects an error and decreases when a successful transmission occurs. REC increases when a receiving node detects an error and decreases for every successful message received. The node states transition is illustrated in Figure 2-22. When the node in the error active state means the node works in the normal condition. Either REC is greater than 127 or TEC is greater than 127, the node transits to error passive state. In the error passive state, the node can transmit and receive messages as well as the error active state, but the node must wait longer before transmitting another message. Only passive error frames can be transmitted. If the REC and TEC are reduced smaller than 128, the node will move back to error active mode. If the TEC is greater than 255, the node transits to the bus off state, which means the node is disconnected to the network. Only to reset the node can make the node transits to the error active state.

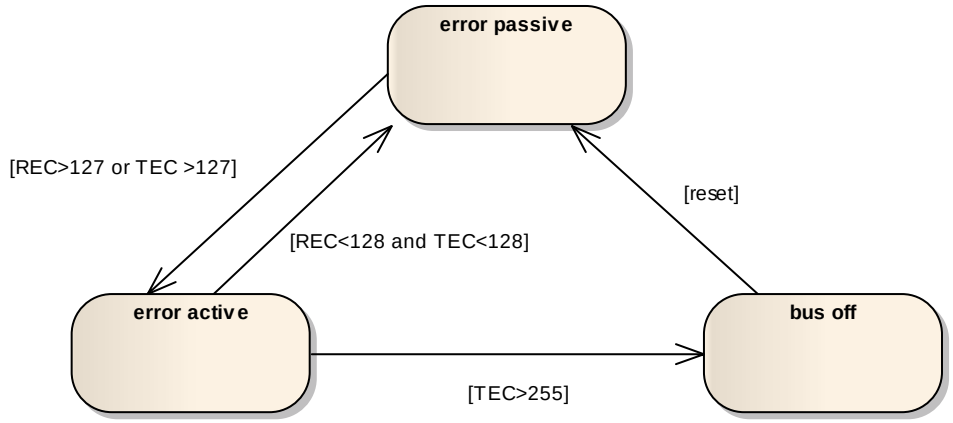


Figure 2-22 a CAN node status

2.4 Inter-task communication

For the portability of the ECU application software, OSEK/VDX and AUTOSAR offers the communication layer. The application software can communicate to each other through the common interface. The communication includes internal and external task communication.

2.4.1 OSEK COM

The OSEK/VDX COM standard supports both intra-ECU task communication and inter-ECU task communication. The standard describes the method to exchange data between different tasks on the same ECU and the tasks on the different ECUs. The internal/external messages are sent by the application but received by the local application and by the application running on the different ECUs through a network. The network of the ECUs connection can be CAN, FlexRay, etc.. (Joseph Lemieux 2001, p125-126).

The OSEK/VDX COM is 5 layer communication model and the ISO/OSI (International Standard Organisation/Open system Interconnections) is 7 layer model. They are shown in Figure 2-23. The application tasks are running on the

application level. The CAN protocol defines the data link layer and the part of the physical layer in the OSI model. The OSEK/VDX COM defines the Interaction layer; it merely defines minimum requirements for the Network Layer to support all features of the Interaction Layer (IL) (OSEK/VDX 2004, p6).

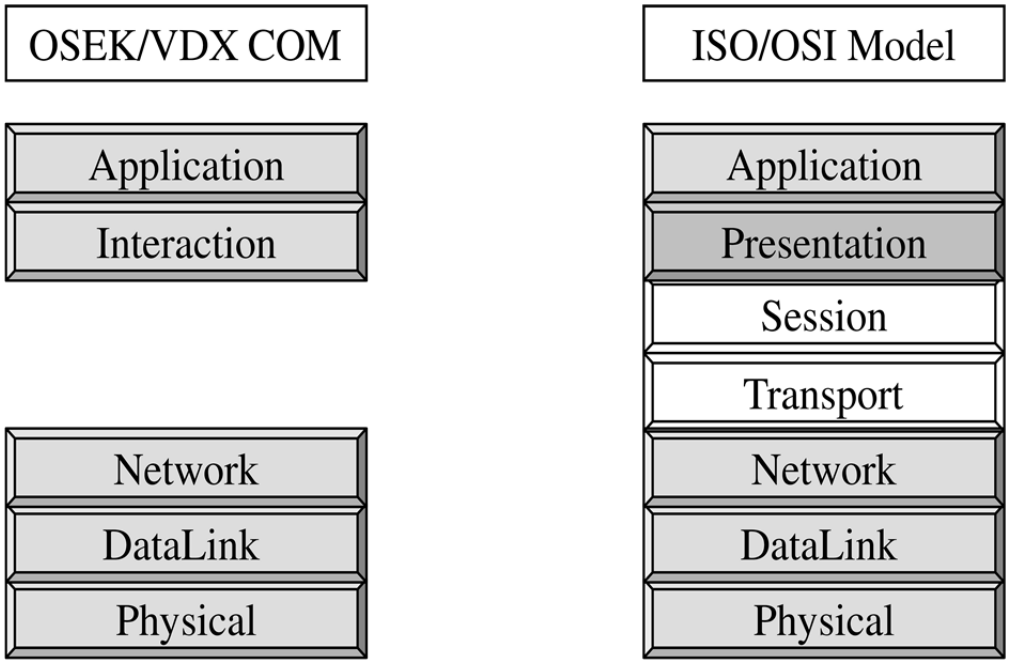


Figure 2-23 OSEK/VDX COM model vs. ISO/OSI model

The interaction layer is shown in Figure 2-24 (OSEK/VDX 2004, p8). The IL defines message (sending or receiving) as message objects. It makes the internal communication message immediately available to the receiver. The external messages (or message) are packed into assigned Interaction Layer Protocol Data Unit (I-PDU). They are passed to underlying layer. The receiving message s pass through underlying layer to the I-PDUs (it contains one or more messages). One I-PDU stores one message, the message is not split across different I-PDUs. Within an I-PDU messages are bit-aligned. The size of a message is specified in bits.

The bit order of a byte in CPU may differ from the order of other CPUs and the order of the byte of the network. The IL makes the bit order same as the order of the local machine. The external message object format is as same as the format of the internal message object. The message object is delivered to the application task by the interface IL offers.

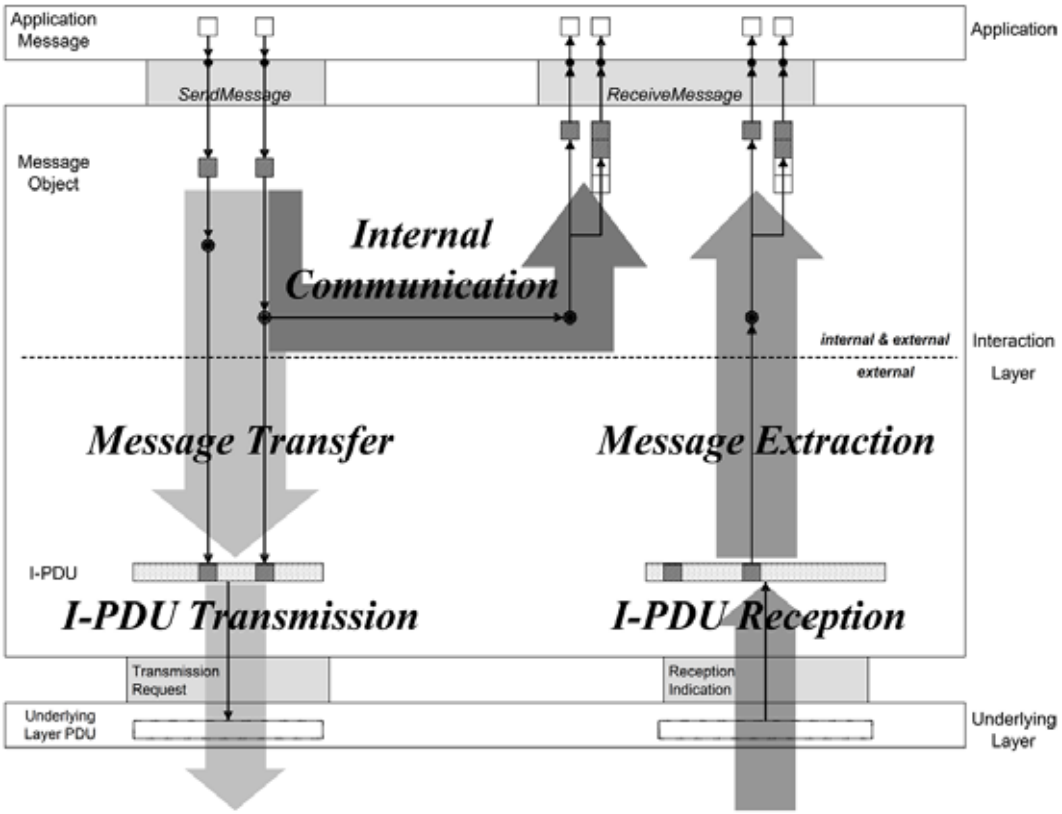


Figure 2-24 message transmission and reception in OSEK/VDX

2.4.2 AUTOSAR COM

AUTOSAR COM layer is the layer between RTE and PDU router. The PDU Router module provides services for routing of I-PDUs between the following modules:(AUTOSAR GbR 2006c, p8)

- communication interface modules (e.g. LINIF, CANIF, and FlexRayIf)
- Transport Protocol modules (e.g. CAN TP, FlexRay TP)

- AUTOSAR Diagnostic Communication Manager (DCM) and Transport Protocol modules (e.g. CAN TP, FlexRay TP)
- AUTOSAR COM and communication interface modules (e.g. LINIF, CANIF, or FlexRayIf) or I-PDU Multiplexer
- PDU Multiplexer and communication interface modules (e.g. LINIF, CANIF, or FlexRayIf)

Figure 2-25 shows the AUTOSAR communication structure.

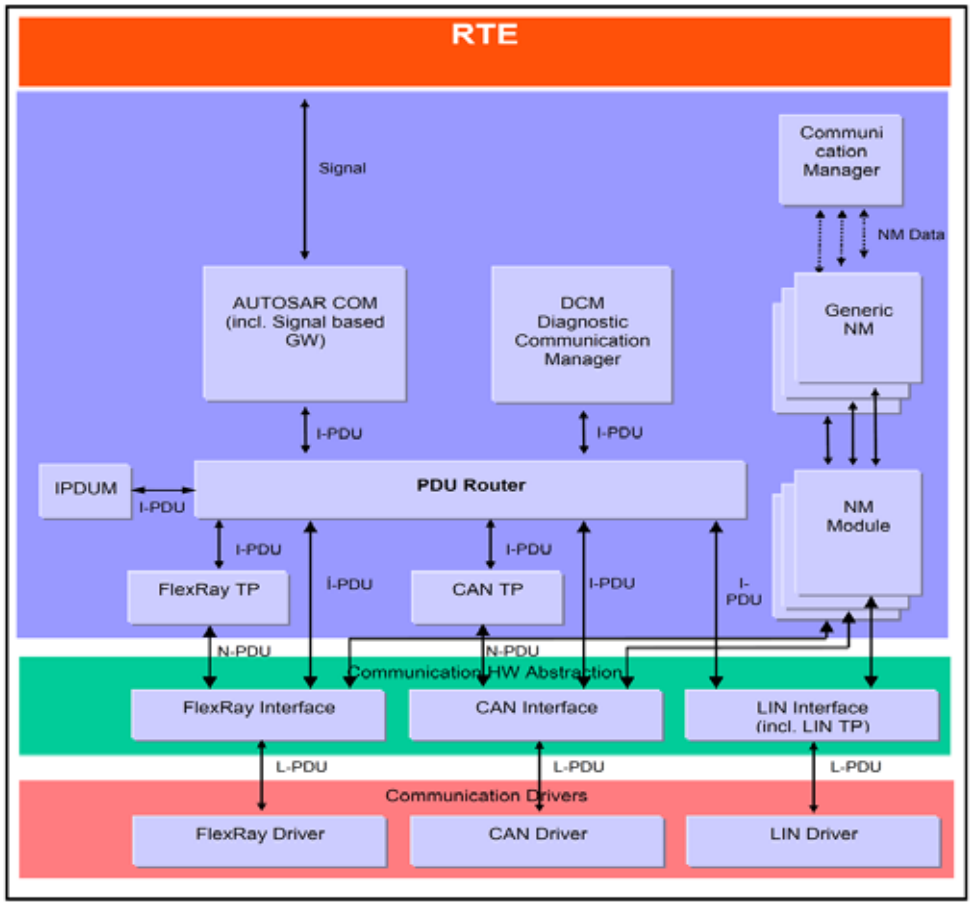


Figure 2-25 Communication Structure

AUTOSAR COM is derived from OSEK/VDX COM (AUTOSAR GbR 2008b, p9). However, AUTOSAR COM provides signal gateway functionality. It forwards signals and signal groups in the one-many manner. Signal and signal

groups are assigned unique static names. The name indicates the destination of the signal. This indication information configure in a table. The signal gateway use this table to find the destination of the signal.

2.5 Event-triggered system vs time-triggered system

In the automotive systems, the activation of ECU functions can either be event-triggered (asynchronous) or time-triggered (synchronous) as well as ECU networks (CAN and FLEXRAY). They all have their own advantages and disadvantages. This section discusses these two systems.

2.5.1 Event-triggered system

For the ECU functions, the event-triggered system activates the function by an event e.g. message received, timer expires, and other function call etc.. The events is possible to happen any time.

CAN is an event-triggered ECU communication network. Event-triggered means that messages are transmitted to signal the occurrence of significant events (e.g., a door has been closed).

In an event-triggered system, the scarce resources (CPU, memory, and network) of real-time systems can be efficiently used. The resources are only used when the event happen. The event-triggered system is easy to be designed, the application runs when the event happens. The new function or new node can be easily integrated to the system e.g. a new ECU can be easily plugged into a CAN network.

Due to the arbitrary nature of the event-triggered system, it is difficult to ensure deadlines are met over load conditions (many events occurring together). For the event-triggered distributed system, such arbitrary and the unpredictable features make the debug and test very difficult. In addition are safety consideration, for example, if the event is missed it could have a safety impact.

2.5.2 Time-triggered system

In the time-triggered system, there is master scheduler which defines the time cycle of the execution. For each execution cycle the scheduler assigns the processor to the tasks or transmits message in the configured time interval, thereby the execution of the task or the message transmission is guaranteed (this feature is ideal for the safety systems) and the resources required are easier estimate and schedule. Another big advantage of the time-triggered system is that it is easy to construct distributed system global states due to the synchronized global clock as discussed at the beginning of section 2.3. It makes the testing and debugging the distributed system easier.

Because all the network transmission or system processing scheduled ahead, it makes the design process very intensive. It causes the future extension of the system to be more difficult. If not enough time intervals are reserved for the future design, the whole system may have to be redesigned.

2.6 ECU calibration, measurement and diagnostics

To debug the distributed system, it is necessary to be able to read the local node variables. For the automotive system, there are some ways to read the memory. This section will describe the protocol to calibrate and measure the ECU variables.

The CAN Calibration Protocol (CCP) is a CAN network based application protocol for calibration and measurement data acquisition of ECUs. The protocol configures the hardware as master/slave structure as shown in Figure 2-26. The master device sends the command to the slave device; slave device gives the response back to master device. Therefore, CCP consists of two message objects:

1. Command Receive Object (CRO) is the message (command) that a master device sends to slave device.
2. Data Transmission Object (DTO) is the message (response) that a slave device replies to the master device.

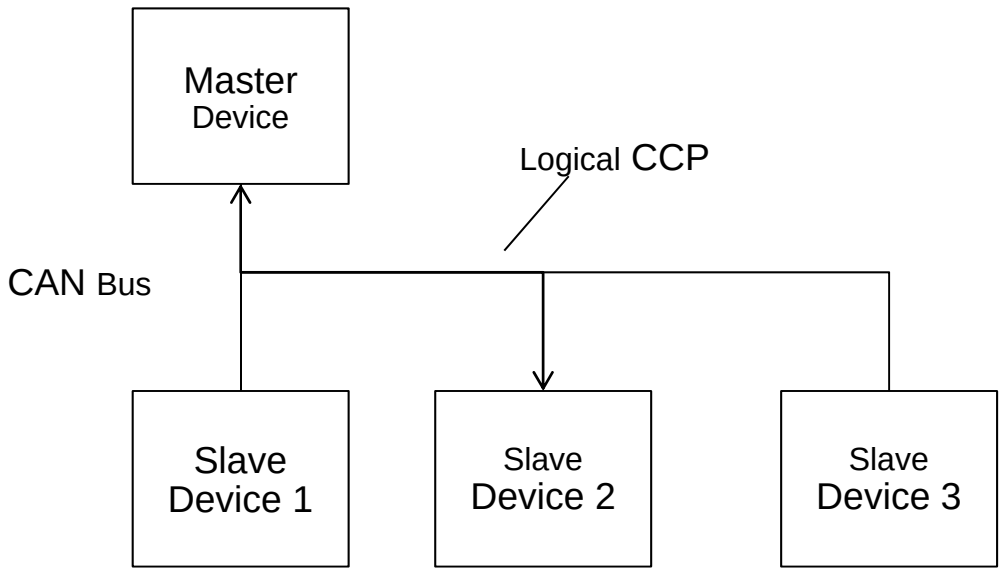


Figure 2-26 CCP master/slave device configuration

Since the CCP protocol is based on the CAN protocol, these two objects are defined in the data field of a CAN data frame. The structure of CRO is illustrated in Figure 2-27. CMD is Command Code which is a byte. It is identifies the command, e.g. 0x01 is CONNECT, the master device sends a connection

command to the slave. CTR is Command Counter, which counts the command message.

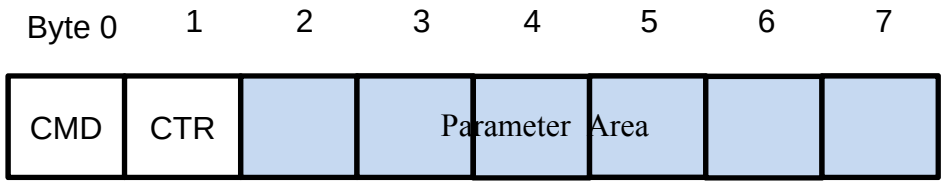


Figure 2-27 CRO structure

DTO includes three types of message:

- **Command Return Message** CRM, if the DTO is sent as an answer to a CRO from the master device. CRM is shown in Figure 2-28. PID is Packet ID, which is used to distinguish between different types of DTOs. The PID 255 is CRM. ERR is the error code. CTR is Command counter as received in CRO with the last command.
- **Event Message**, if the DTO reports internal slave status changes in order to invoke error recovery or other services. The structure of Event Message is same to CRM, except its PID is 254.
- **Data Acquisition Message (DAQ)**, if the DTO contains measurement data. It contains the data in the memory of the ECU. The structure of a Data Acquisition Message is illustrated in Figure 2-29. It only has PID field, the rest are data area. The range of the PID is from 0 to 253.

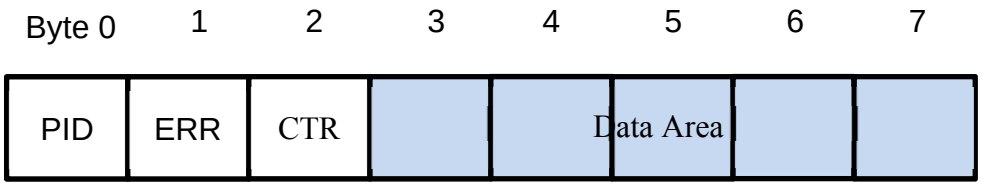


Figure 2-28 CRM structure

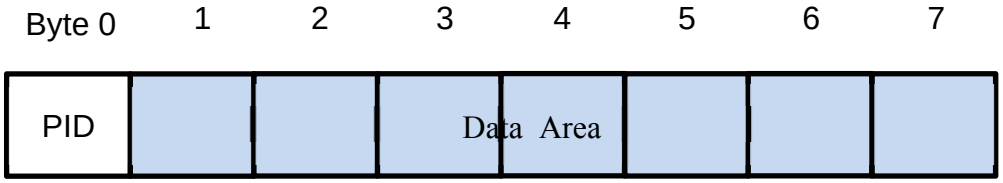


Figure 2-29 Data Acquisition Message structure

The DAQ contains data corresponding to an Object Description Table (ODT) that maps the memory address of the ECU. Figure 2-30 illustrates an ODT. Each address points the value stored in the memory of the ECU. ODT is assigned a unique Packet Id PID to identify the corresponding DAQ (DAQ-DTO). The contents of each element defined in a ODT are transferred into a DAQ-DTO to be sent to the master device. Multiple ODTs form a DAQ list debugging (H.Kleinknecht 1999;Rainer Zaiser 2011).

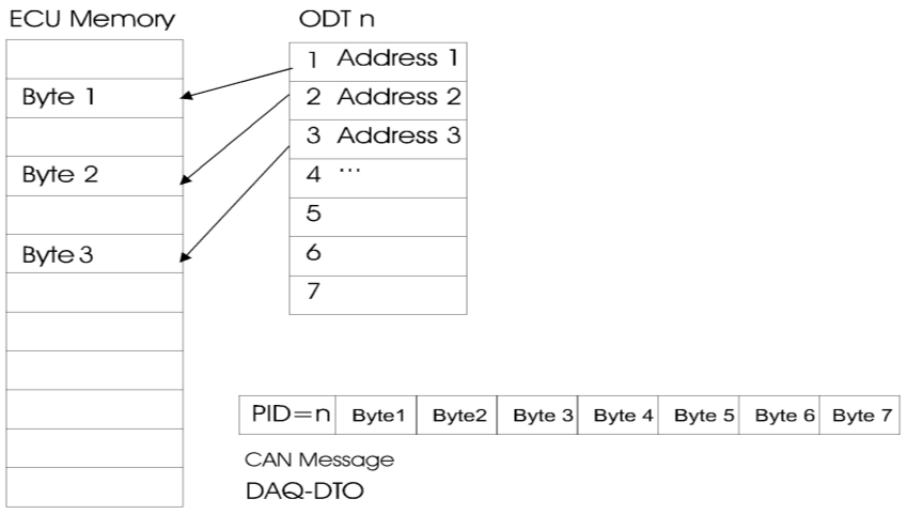


Figure 2-30 Object Descriptor Table

2.7 Conclusion

This chapter has introduced automotive application development. The application task running on the ECU is managed by the automotive standard OS. The most popular one is OSEK/VDX OS and AUTOSAR OS is becoming more widely used. The application task needs to interact with other tasks. These tasks can either run on the same processor (internal communication) or run on a different processor (external communication). For the external communication tasks, the message can be sent through different networks, e.g. CAN, FlexRay, etc. but FlexRay networks has a global time, all nodes are synchronized by this global time. However, for the predominant CAN automotive network system, there is no such mechanism to synchronize all nodes. That's also the difficulty for debugging the distributed automotive system. Therefore, the research only focuses on the CAN based network. For the portability and compatibilities of the tasks, the communication layer services were developed by the OSEK/VDX and AUTOSAR. Finally, the CCP protocol which measures and calibrates ECU was introduced. The measurement and calibration is essential successful integration and deployment of ECUs.

References

AUTOSAR GbR. requirements of RTE. 7-12-2006a.

AUTOSAR GbR. software component template. 6-26-2006b.

AUTOSAR GbR. Specification of PDU Router. 6-26-2006c.

AUTOSAR GbR. Layered Software Architecture. 2-14-2008a.

AUTOSAR GbR. Specification of Communication. 2-13-2008b.

AUTOSAR GbR. Specification of Operating System. 6-23-2008c.

AUTOSAR GbR. Specification of RTE. 9-22-2010.

BOSCH. CAN Specification. 1991. Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart.

Brendan Jackman. Basic Concepts. An overview of the distinguishing features of the CAN network. 2004a. Waterford Institute of Technology, Ireland.

Brendan Jackman. CAN Frame Formats. 2004b. Waterford Institute of Technology, Ireland.

Darren Buttle. What is an RTE. Introduction to AUTOSAR for RTE users. 12-5-2005.

H.Kleinknecht. CAN Calibration Protocol Version 2.1. 2-18-1999.

Joseph Lemieux. Programming in the OSEK/VDX Environment. Berney Williams, Robert Ward, Rita Sooby, and Michelle O'Neal. 2001. CMP Books.

LiveDevices Ltd. RTA-RTE User Guide. 2004.

Nicolas Navet & Françoise Simonot-Lion. Automotive Embedded Systems Handbook. 2009. Taylor & Francis Group, LLC.

OSEK. OSEK/VDX Operating System Specification 2.2.3. 2-17-2005.

OSEK/VDX. OSEK/VDX Communication. 7-20-2004.

Rainer Zaiser. CCP. A CAN Protocol for Calibration and Measurement Data Acquisition. 2011. Vector Informatik GmbH Friolzheimer Strasse 6 70499 Stuttgart,Germany.

Richard Zurawski. networked embedded systems. 2009. Taylor & Francis Group, LLC.

Robert Warschofsky. AUTOSAR Software Architecture. 2011. Hasso-Plattner-Institute fuer Softwaresystemtechnik.

Simon Fuerst & BMW AUTOSAR An open standardized software architecture for the automotive industry, *In 1st AUTOSAR Open Conference & 8th AUTOSAR Premium Member Conference*.

Stefan Bunzel. Overview on AUTOSAR Cooperation. 5-13-2010. Tokyo, Japan, 2nd AUTOSAR Open Conference.

Chapter 3 Automotive Software Testing

3.1 Introduction

Software testing is a very important part of the software development process. It will cross entire development life cycle. If not enough testing has been done before the software is delivered, it can cause many problems e.g. damage the reputation of the company, bringing the danger to the customers and so on. A lot of money is lost every year with vehicle recalls. However it is impossible to find all the bugs during the development life cycle. For example Microsoft will update windows after it has been published. “The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.” (Patton 2005)

In the automotive industry, software testing is very important issue, because nowadays all cars are controlled by the software, even some safety features. Before the software can be used in a production; it must have been passed a huge number of testing, even though it cannot be guaranteed that all possible test cases can be tested.

This chapter combines the general software test and automotive software test together, to describe the steps for automotive software testing, what kind of test should apply to each development phase, and the industry tools are used for the ECU integration.

3.2 V model

The V model was developed in the 1980s (National ITS Architecture Team 2007), as the German industry standard. It is the most predominant development cycle in the automotive industry (Schaeuffele and Zurawka 2005, p24).

The V model was developed from waterfall software development methods (Schaeuffele & Zurawka 2005), and they have common disadvantages and advantages; they are good for the project that is defined well, the requirements are fully understood. Because of its sequential nature, it is not flexible; it supposed to give a complete system at the end, but if the requirements change during the development, it is difficult to go back (A.Al-Ashaab et al. 2009;Pressman 2001).

Figure 3-1 Overview of the core process for the development of electronic systems and software

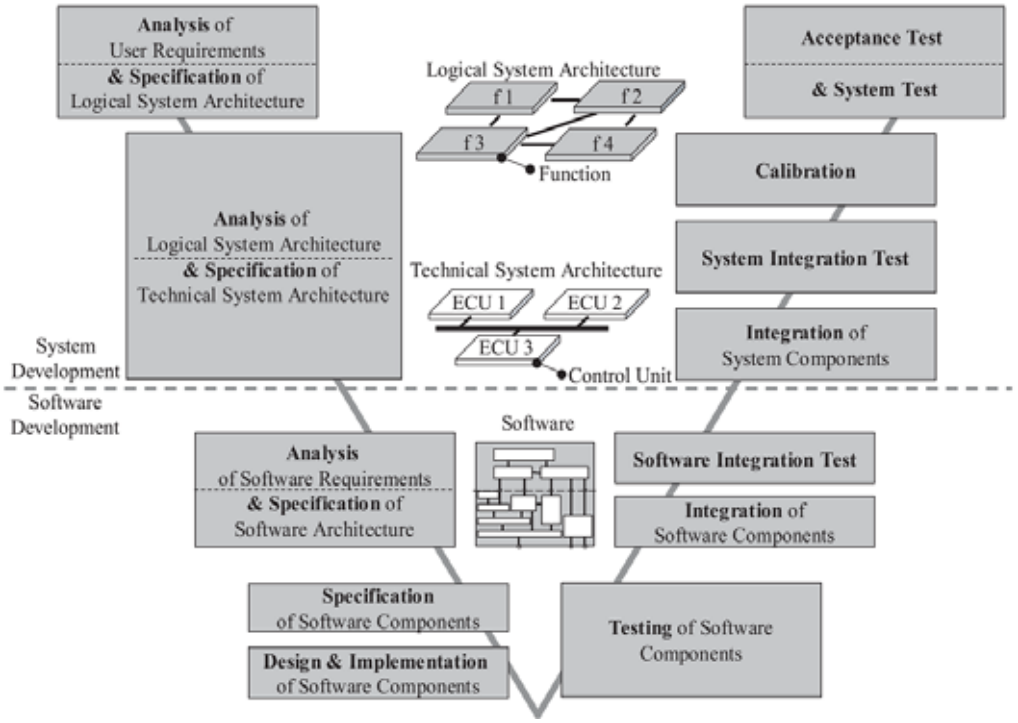


Figure 3-1 (Schaeuffele & Zurawka 2005, p24) is an overview of the V model

- **Analysis of user requirements and specification of logical system architecture**

  Developers have meeting with users, to find the requirements from the users. The user requirements are analysed. The uses case is defined. The system to be developed depends on these use cases, also the acceptance

tests are defined. The requirements are linked and the technical system architecture is designed. These can be expressed and modelled by using diagrams (block diagram, UML etc.).

Logical system requirements are formulated based on two different perspectives: functional and non-functional requirements.

Define the logical system architecture based on the requirements that have been found. Logical system architecture is the model of the function network, function interface and the communication among the functions. It does not involve any technical implementation.

Decomposition of system function is used to determine the system components, interface and functions. Function network describes the relationship among the functions (e.g. dependency). The communication networks describe how the functions communicate with each other (e.g. CAN, LIN and FlexRay). The functions are grouped into components.

- **Analysis of the logical system architecture and specification of technical system architecture**

  The specification of the technical system is based on the logical system architecture. To decide use what hardware (ECU) implement which function or functions by considering constrains of ECU. Because some ECU may not be suitable for the function, the ECU is better to implement the function than the others or the price of ECUs ect.. The software requirements are defined.

- **Analysis of software requirements and specification of software architecture**

  Define software boundaries and interface. The software boundaries which items are part of the software system, and which items belong to the periphery or environment. The software interface defines includes two types interface, on board and off board. On board interface includes set-point generators, sensors and actuators. Off board interface include downloading and debugging tool, flash programming tool, diagnostic tool and network development tool .

- **Specification of software components**

  This step involves modelling software components, the implementation detail is ignored. There are three types of model to be specified. They are the data model, the behavioural model and the real-time model. The data model defines the data to be processed by the software. The behavioural model specifies the dynamic structure of software components. It includes specification of data flow and control flow. Specification of data flow is a processing flow of the data (input to output) among the software components. It describes the paths of data transfer between software components. Specification of control flow describes the control of the instructions' execution. There are four control structures: sequence, branching, repetition (iteration) and call. The real-time model defines the real-time requirements of the task that assign a process that implements software component, such as deadlines for event handling.

- **Design, implementation, and testing of software components**

  The design phase must define all specific implementation for the data, behaviour and real-time model of the software components. Consideration

of the requested non-functional product properties (hardware cost, reliability etc..). Design and implementation of the three models is based on the specification of the software components. Every implementation is tested by a corresponding test method during the development.

- **Integration of software components and software integration tests**
  After all software components are completed and have passed the corresponding test e.g. unit test, they are integrated into a software system and integration testing can start.

- **Integration of system components and system integration tests**
  This step installs all programs into the ECUs. All of these ECUs are connected to the other electronic devices (setpoint generators, sensors, and actuators).

- **Calibration**
  This involves setting data point to give optimum system performance.

- **System and acceptance test**
  This step checks that if the system satisfies the user requirements.

3.2.1 Multiple V models

As in other transportation industries (train and aerospace), the automotive system is a big project. It can't be built directly after it is designed. The model of the system is built on a PC and is a simulation of the system. If the model is correct, then the code is generated and embedded in a prototype. The hardware of the prototype will be gradually replaced by the real hardware. Eventually the final product will be formed in this way. The reason for building the model and prototype is because changing a prototype is easier and cheaper than changing the final product. Also at the beginning the user requirements are very hard to be fully

understood. The model can help the developers to better understand these requirements. It also can be shown to the user to check if the model satisfies their requirements. The validation of the system happens on the early stage and can help reduce the risk.

In the multiple V models, each V development cycle develop the same required functionality. But they are developed in the different physical versions of the same system. This means the same functionality can be tested for the model as well as for the prototype and the final product. The difference is the executing environment.

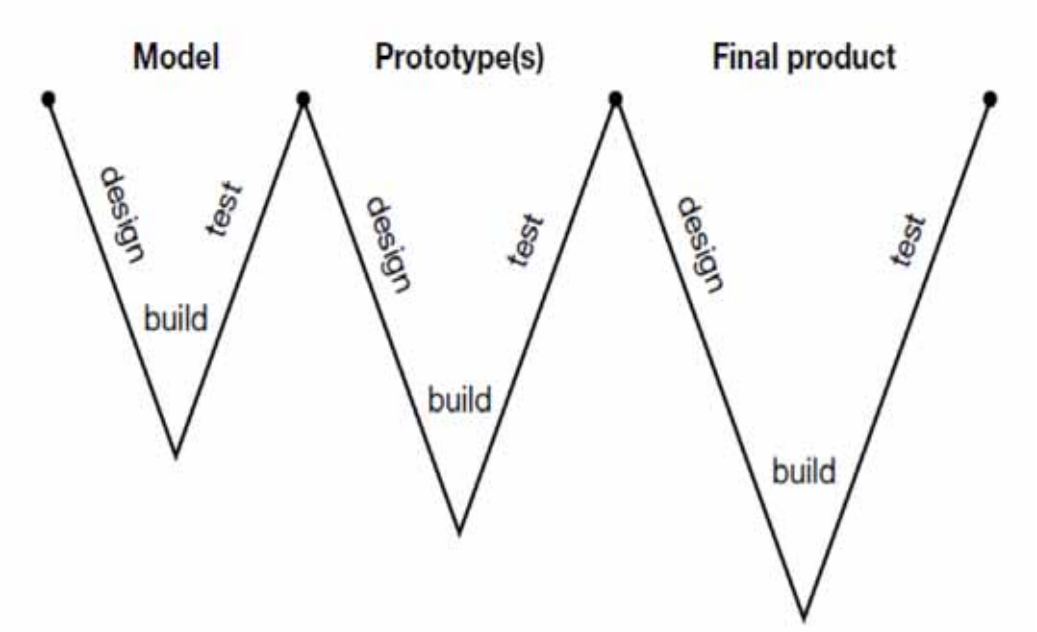


Figure 3-2 Multiple V development life cycle

In a single V model method, the testing starts after all implementation is done. It is not on adequate approach in today's iterative software processes. It may work for some small, simple and well understood project, however nowadays software gets bigger and more complex each year.

Automotive software development involves both software and hardware, often developed independently and in parallel. It is big risk to find that the software and hardware don't work together at a very late stage of the development. The iterative development method allows frequent communication, integration and testing between them. The system is built little by little. After one small part of the system functionality is built (hardware and software) and it is successfully integrated and tested, the next small part of functionality is built and so on.

To develop some really large system a decomposition of the system is necessary. The works are assigned to groups. Each group develops part of the system in parallel. The multiple V process is applied to every group. The integration of the different components happened many times in the development process. The early stage integration is based on different component models or lab hardware. In the end the final product is integrated together when all the components have been fully developed.

3.2.2 Nested V model

The multiple V process does not address the decomposition of complex systems which are very common in the automotive industry. A high level process is needed to decompose the system. Also at the end, a process is required to recompose the system. As a matter of fact, a single V process can be applied to achieve such process requirements. This is shown in Figure 3-3. The left side of the V process is to decompose the complex system into the components. The middle of V process is the development of these components in parallel. On the right side of the V is the integration of all the components.

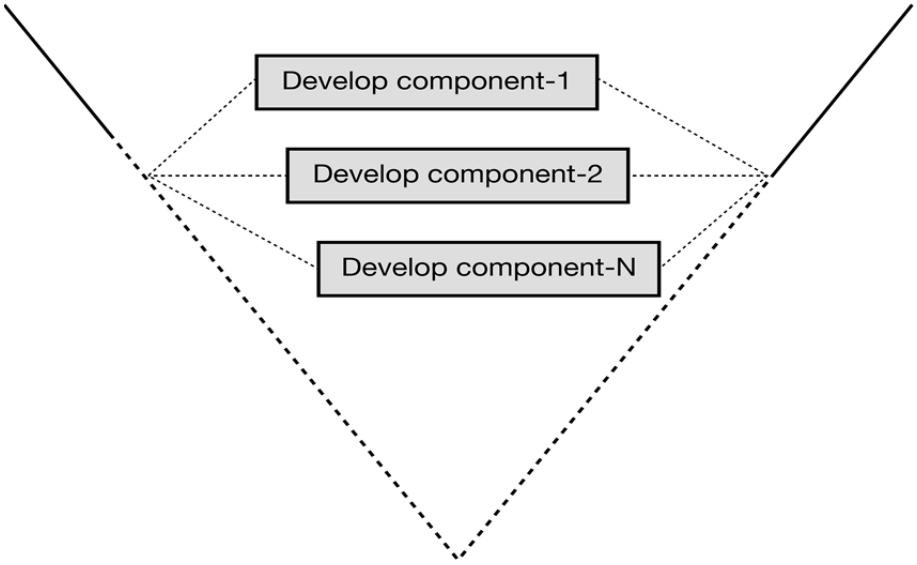


Figure 3-3 parallel development phases in V model

After the system is decomposed, since each component is not as complex as before, a multiple V model can be applied to the component development. The whole development process is like many multiple Vs nested in a V model as Figure 3-4 illustrates.

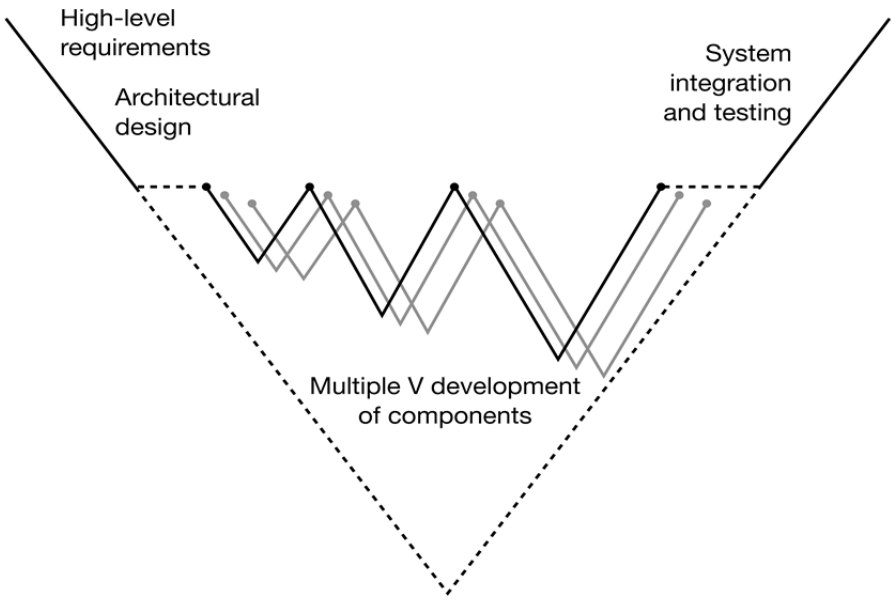


Figure 3-4 system decomposition and development using nested and multiple V models

By using a nested V model, everything related to system testing can be addressed at the right place and level as illustrated in Figure 3-5.

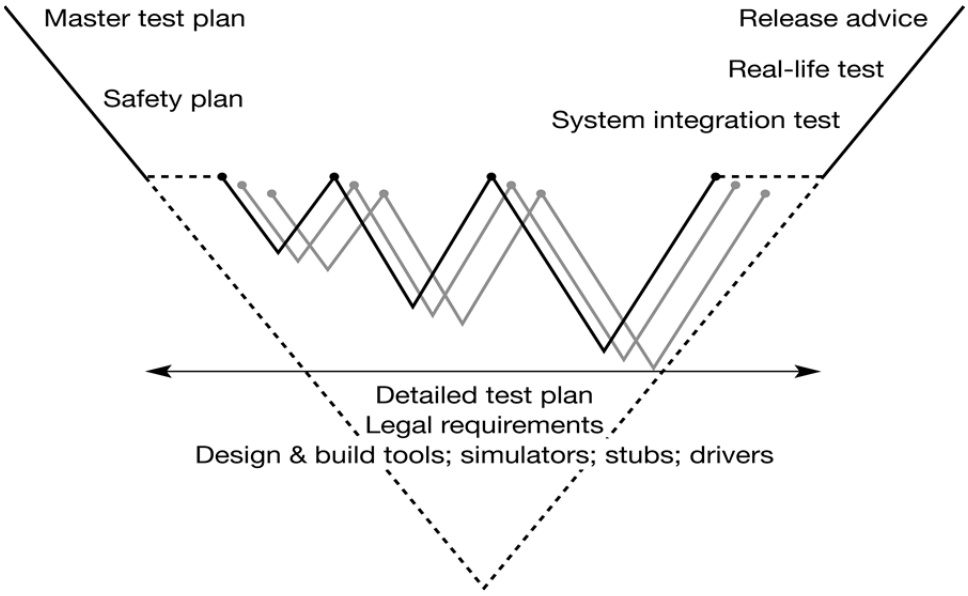


Figure 3-5 Higher-level test issues in the nested multiple V model

3.3 Test planning

A modern car contains a lot of hardware components and the associated software control systems are extremely large. During the development many tests should be done in the different stages; some of them test performance, others test component integration, others test the user-friendlessness etc.. Lots of complex situation appear, so plan are needed to control the testing process.

For complex systems, making a master plan can give an overview of the test control process. There are two fundamental aspects to the master test plan: test type and test level.

A system can be tested from different points of view; functionality, user friendlessness, performance, etc. These attributes are essential for software quality assurance. Some of the quality attributes are related so we can define them in the

test type. “A test type is a group of activities with the aim of evaluating a system on a set of related quality attributes.”(Broekman and Notenboom 2003, p33). Test types state what is going to be tested (and what is not). It gives a boundary for the tester to test.

Table 3-1 lists some common test types (Broekman & Notenboom 2003, p34).

Test type	Description	Quality characteristics included
Functionality	Testing functional behaviour (includes dealing with input errors)	Functionality
Interfaces	Testing interaction with other systems	Connectivity
Load and stress	Allowing large quantities of events and numbers to be processed	Continuity, performance
Support (manual)	Providing the expected support in the system’s intended environment (such as matching with the user manual procedures)	Suitability
Production	Test production procedures	Operability, continuity
Recovery	Testing recovery and restart facilities	Recoverability
Regression	Testing whether all components function correctly after the system has been changed	All
Security	Testing security	Security
Standards	Testing compliance to standards	Security, user-friendliness
Resources	Measuring the required amount of resources (memory, data communication, power, …)	Efficiency

Table 3-1 Common Test Types

“A test level is a group of activities that is organized and managed as an entity.” (Broekman & Notenboom 2003, p34). Test levels states who is going to perform the testing and when. The test level can be defined as high-level tests and low-level tests. The high-level tests are tests on the integrated system or subsystem, they are more black-box oriented. The low-level tests are tests on isolated components, they are more white-box oriented. In the nested V model development process, the low-level test is at left side of V, the high-level test is at right side of V. Table 3-2 list test level (Broekman & Notenboom 2003, p35).

Test level	Level	Environment	Purpose
Hardware unit test	Low	Laboratory	Testing the behaviour of hardware component in isolation
Hardware integration test	Low	Laboratory	Testing hardware connections and protocols
Model in the loop	High/low	Simulation models	Proof of concept; testing control laws; design optimization
Software unit test, host/target test	Low	Laboratory, host + target processor	Testing the behaviour of software components in isolation
Software integration test	Low	Laboratory, host + target processor	Testing interaction between software components
Hardware/software integration test	High	Laboratory, host + target processor	Testing interaction between hardware and software components
System test	High	Simulated real life	Testing that the system works as specified
Acceptance test	High	Simulated real life	Testing that the system fulfils its purpose for the user/customer
Field test	High	Real life	Testing that the system keeps working under real life conditions.

Table 3-2 Test levels

At the begging of the project, a master plan (Figure 3-6) needs to be drawn up which defines the tasks, responsibilities, and boundaries for each test level. A master test plan describes how to combine the test type and test level together; test type is what has to be tested and test level is who is going to perform the test. Three areas are of main interest for the master test plan:

- Test strategic choices – what to test and how thorough;
- Allocation of scarce resources;
- Communication between the disciplines involved.

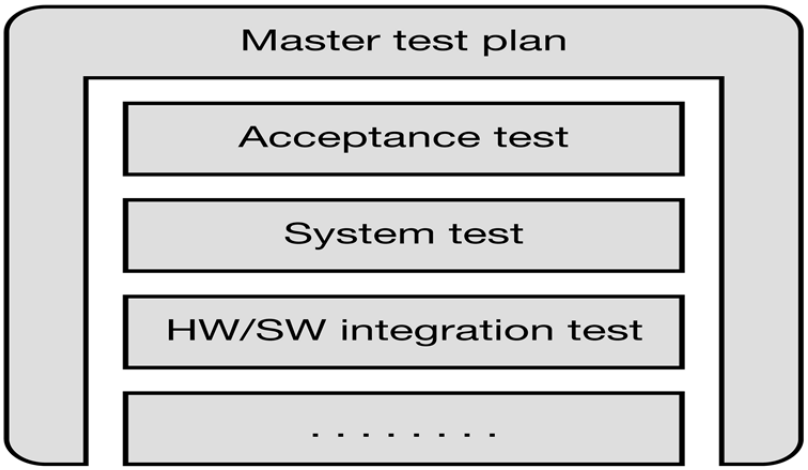


Figure 3-6 Master test plan

After the master test plan has been made, the detailed plan of each test level is based on the master test plan. A typical test plan should include the following:

1. application(s)/system(s) to be tested
2. testing objectives and their rationale (risk and requirements)
3. scope and limitations of the test plan
4. sources of business expertise for test planning and execution
5. source of development expertise for test planning and execution
6. sources of test data
7. test environments and their management
8. testing strategy
9. <Repeated> testing details for each development phase
   (a) development phase
   (b) how can you tell when you are ready to start testing?
   (c) how can you tell when you are finished testing?
   (d) <Draft> test cases list (ID, title, and brief description)
   (e) <Draft> test case writing schedule
   (f)<Draft> test case execution schedule
   (g) <Draft> test case execution results analysis and reporting schedule
10. <Draft> overall testing schedule

<Repeated> means that you should expect to repeat this item and all sub-items for as many times as there are development phases.

<Draft> means that at the time the test plan is first written, there is insufficient information from the development activities to fully document the <Draft> items.

3.4 Verification and Validation

There are two types of fault in software products. They are specification faults and implementation faults. The specification faults are predominant in most projects from the research result (Michael Schneider et al. 1992), so the V model differentiates between verification and validation.

Verification checks if the software satisfies the specification (Patton 2005). Was the software built right?

Validation checks if the software satisfies the user requirements (Patton 2005). Was the right software built?

The V model separated four test steps:

- Component test versus component specification.
- Integration test versus specification of the technical system architecture.
- System test versus specification of the logical system architecture.
- Acceptance test versus user requirements.

Figure 3-7 (Tian 2005, p31) shows the verification and validation activities associated with V model

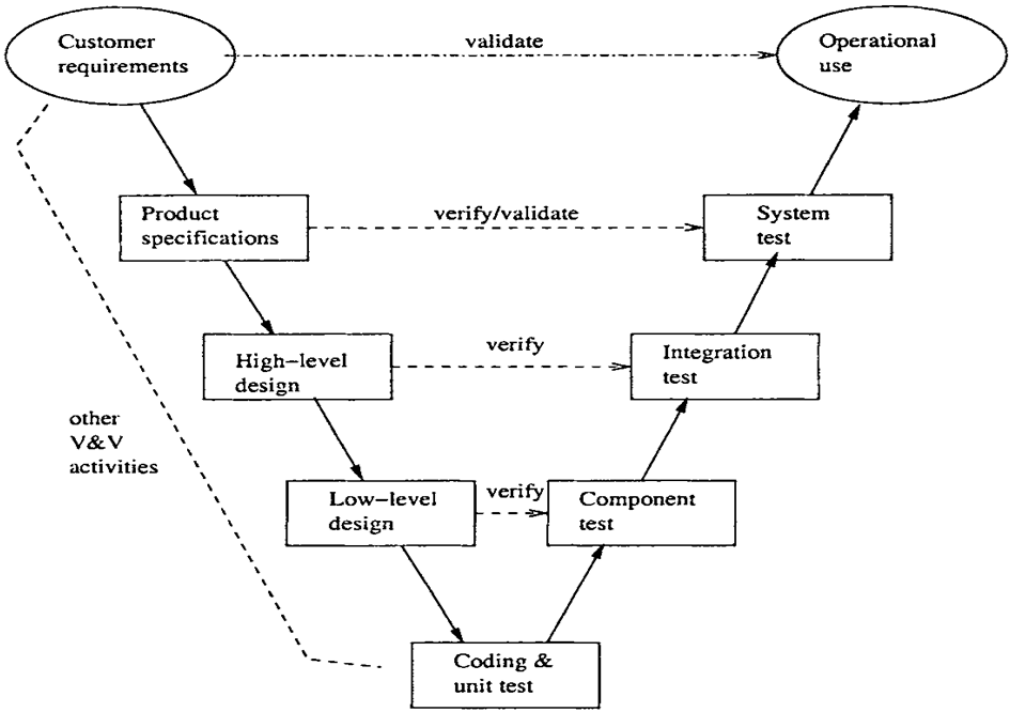


Figure 3-7 Verification and validation activities associated with the V model (Tian 2005, p204)

3.4.1 Verification

There are two methods for the software verification: static and dynamic method.

3.4.1.1 Static testing

Static testing is “a process of evaluating a system or component without executing the test object.” (Broekman & Notenboom 2003, p331). It examines the documentation that has been produced during the development. Static test is the least expensive testing, giving a big opportunity to reduce defects that have been written in the documentation.

There are three static testing techniques to review documents' content.

- Desk checking
- Inspections
- Walk troughs

Desk checking is done by the author. He (she) check his (her) own documents. “Desk checking involves first running a spellchecker, grammar checker, syntax checker, or whatever tools are available to clean up the cosmetic appearance of the document. Then, the author slowly reviews the document trying to look for inconsistencies and incompleteness. Problems detected in the contents should be corrected directly by the author with the possible advice of the project manager and other experts on the project. Once all corrections are made, the cosmetic testing is rerun to catch and correct all spelling, grammar, and punctuation errors introduced by the content corrections.” (Everett and McLeod 2007,p 97).

Inspections require more people to check the documents. The inspectors are more senior members of the team. The document is read by the inspectors who discover the content problems. When they read the document, it is better to avoid the author, because human tendency is for the author may to influence the reviewer. During the reviewing the inspector should record any problems that they see. After they can have the discussion with project manager or someone who is senior in the project to correct the problems.

“The walk-through is a scheduled meeting with a facilitator, the document author, and an audience of senior technical staff and possibly business staff. The author must scrub the document for cosmetic errors and send the document to all participants in advance of the meeting. The participants read the document and formulate questions about the document contents based on their own knowledge of the new system or application. At the appointed time, the author presents his or her document to the walk-through meeting. The facilitator becomes the clearinghouse for questions by the audience and answers by the author. The facilitator also ensures that the questions are posed in an objective, nonthreatening

way. The walk-through facilitator documents all suspected content problems and author responses for later resolution by the project manager in a manner similar to the inspection resolutions" (Everett & McLeod 2007, p98).

All the techniques previously introduced are the review techniques. But code review only can find spelling or syntax errors; it is not very effective for logic errors. So the static code analysis is another important part in the static test. There are two main components: building a model and analysis algorithm.

3.4.1.2 Dynamic test

Dynamic test runs the program as a customer would. There are two test levels: unit test and integration test, and two ways for testers to approach the test: black box testing and white box testing.

3.4.1.2.1 Black box test

In black box testing the tester does not need to know how the software works, it checks if the software does what it is supposed to do. The input and the output is the main thing to test. The process is ignored. For example, to test a vending machine, the tester puts the coin to the machine, select an item and see if the machine gives the right item. The tester does not care how the machine performs the task.

3.4.1.2.2 White box test

To do white box testing, the tester must have access to the program's code and be able to observe the execution trace. The whole execution of the program is monitored. For example, the vending machine will be disassembled; all operations will be monitored from the time the coin has been put in to when the item gets out.

3.4.1.2.3 Component (Unit) test

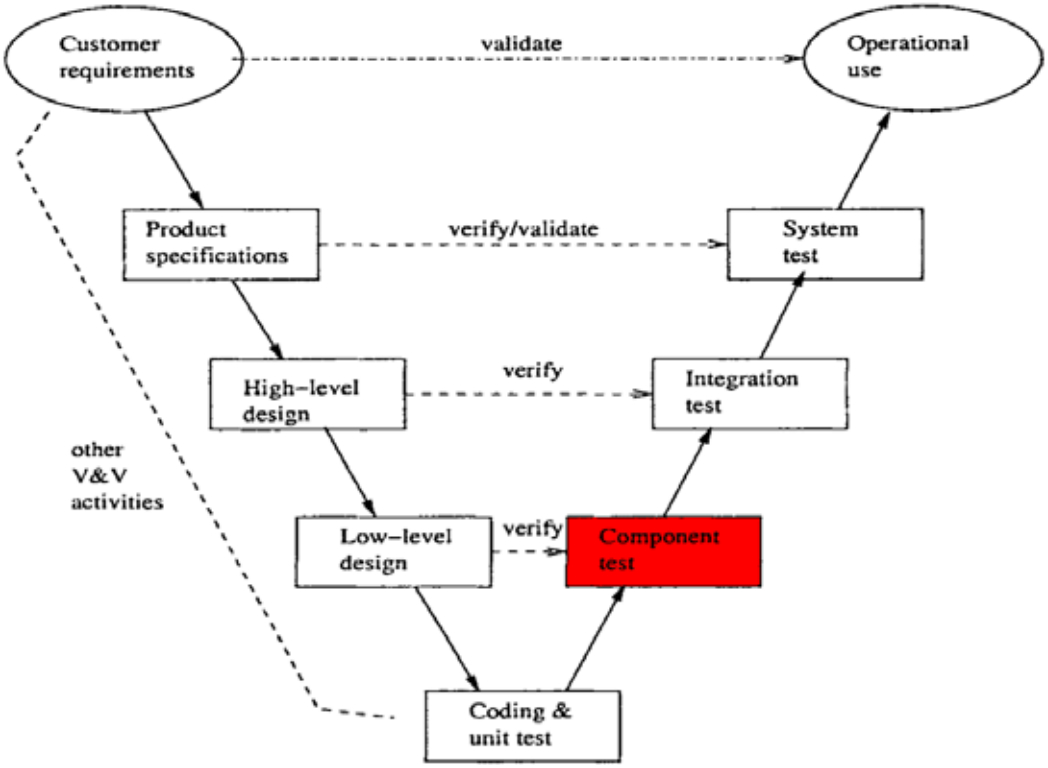


Figure 3-8 V model (Component test)

"The tests that are conducted on the module software are called unit tests or module tests." (Everett & McLeod 2007, p53) The unit test normally happens at the function level. A unit test should apply to each function as they are developed. It makes sure the individual function is working before integrating them together.

Procedure to implement unit testing (Dasso and Funes 2006, p76):

1. *Prepare test environment.*
2. *Define input domain based on requirements and use cases.*
3. *Define, for every input, expected output based on requirements and use cases.*
4. *Implement components to be tested.*
5. *Group unit tests in collections of components.*
6. *Implement unit tests.*

7. *Execute unit testing.*
8. *Fix component tested, if there is an error.*
9. *Execute step 8 while any error remains.*

Unit testing consists basically of (Dasso & Funes 2006, p76):

1. *Variable initialisation, including database population.*
2. *Business rules or input functions are applied.*
3. *Destruction of variables, including the cleaning up of data input to data base.*
4. *Comparison between results of applied function with expected results, failing in cases where they differ.*

3.4.1.2.4 Hardware in the loop (HIL)

HIL is a test level where real hardware is used and tested in a simulated environment. (Broekman & Notenboom 2003, p329) In the real world, the ECUs connect to actuators or sensors. But during the development, it is not very convenient to establish these connections all the time. Some companies produce simulated actuators and sensors to help testing the real ECUs. Vector is one of the companies. One of their products called VT system that is connected to the ECU instead of the actuators or sensors. Figure 3-9 (Vector Informatik GmbH 2010a) shows a block diagram that Vector VT system tests an ECU.

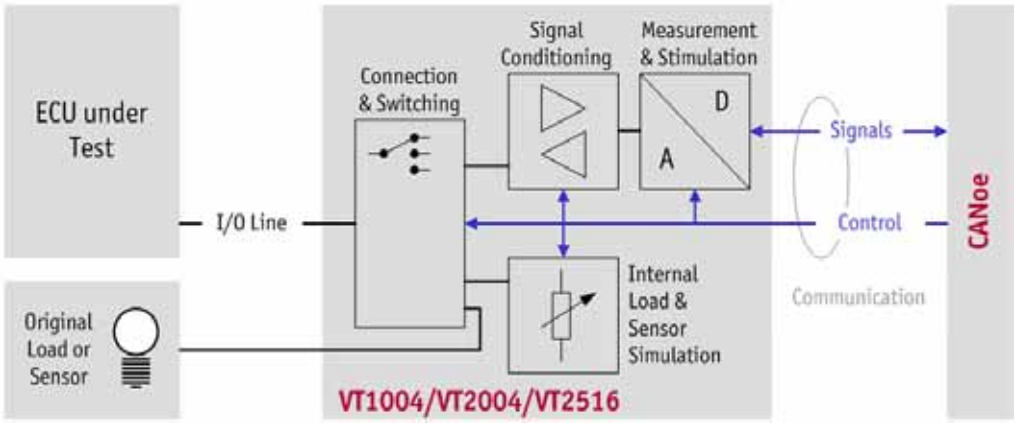


Figure 3-9 Vector VT system

Benefits of HIL Simulation

- Systematic and Reproducible
  HIL simulation is the method used to test the functions, system integration, and communication of electronic control units (ECUs). The ECUs can be applied in vehicle, aerospace, machine tools etc. All the parts (sensor, actuator, etc.) that ECUs connect to are simulated in HIL. HIL is very systematic and safe, even when critical thresholds are exceeded. The main purpose of HIL is to detect errors (e.g. unconnected sensor etc.). If the error has been detected, then this error can be produced again.

- Improving ECU Software Quality
  HIL simulation helps to improve quality at an early development stage. A major Japanese automobile manufacturer states that HIL simulation finds 90% of ECU errors, and almost all the errors can be found before the calibration phase (dSPACE 2009, p102). This shortens the time to market and avoids recall campaigns that damage a company's image. The investments made in HIL systems and in developing tests have usually paid off after only a few months.

The following example is about HIL for a Three-Wheeler Scooter (dSPACE GmbH 2007).

"Piaggio developed the three-wheeler scooter MP3 with two front wheels. The innovative, electronically controlled locking system keeps the vehicle upright without using the usual central stand. The complete system of networked electronic control units (ECU) was tested by ELASIS using a dSPACE hardware-in-the-loop (HIL) simulator.

Figure 3-10 Piaggio MP3 scooter

The new three wheeler scooter MP3 is better road holding in whatever grip conditions and on bad surface roads. It has a parallelogram suspension anchored to the frame that allows a tilt angle of up to 40°. The locking mechanism for the front suspension mainly consists of the NST (Nodo Stazionamento, Locking Mechanism Control Unit) and the engine control unit NCM (Nodo Controllo Motore). The implementation of the NST is feasible only if the electronic control unit (ECU) which controls it is connected to the NCM via a CAN network."

The new locking system NST allows “easy parking” without the kickstand; When the driver pushes the lock request lever, the lock conditions have to be simultaneously verified:

- Vehicle speed below a threshold which is a function of vehicle deceleration
- Throttle closed and engine speed under a threshold

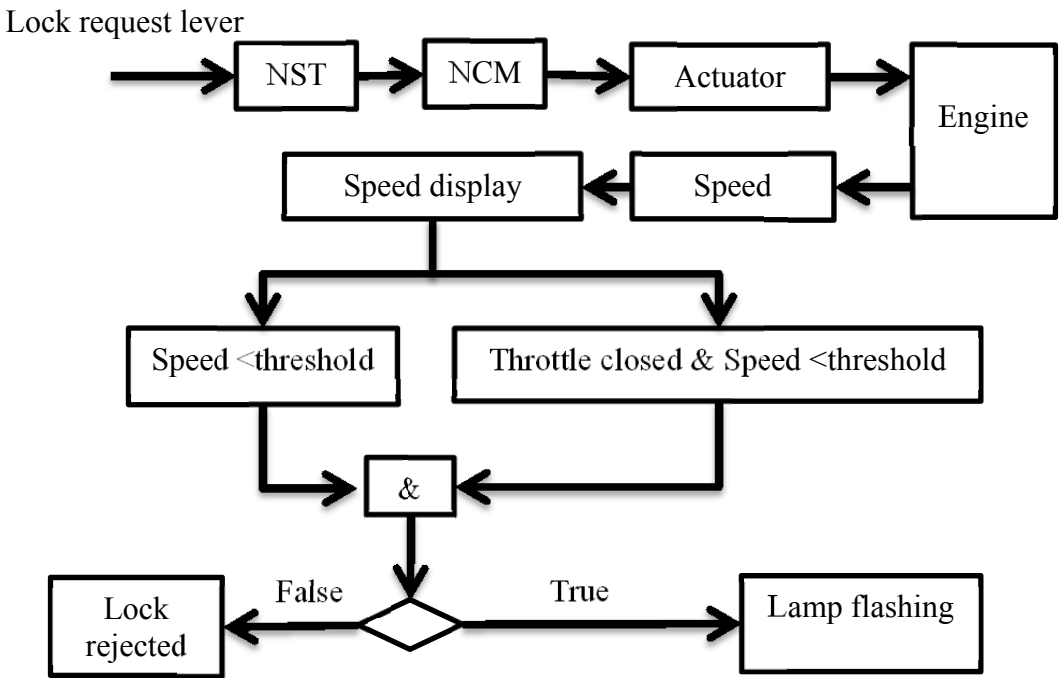


Figure 3-11 “easy parking” system block diagram

If these conditions are not reached after a certain time span, the lock request is rejected. If the locking conditions are true, a lamp on the dashboard starts flashing and is lit permanently when the suspension is locked. When the driver is on the scooter, the suspension is unlocked on the driver’s request and, for safety reasons, if one of the following conditions is verified:

- Engine speed above a threshold which assures that the clutch is closed
- Vehicle speed above a threshold

To test the NST thoroughly, needs a lot of severe testing conditions that are difficult or even dangerous to reach, like cornering sharply or braking at top speed on rain-soaked surfaces. Moreover, it is almost impossible to generate exactly the same testing condition twice. ELASIS tested the NST and the NCM simultaneously on the CAN network. The model of the engine runs in real time to verify correct control system integration on the CAN network. The simulation therefore had to provide a short turn-around time. They also needed a test platform with closed-loop simulation, the facility for test automation, and Fault Insertion Unit (FIU) capabilities. To make sure the locking mechanism will be reliable even if other components fail, FIU is very important. Having this in mind and working towards extending the same development platform for different ECUs, ELASIS selected a dSPACE Simulator Mid-Size as real-time hardware. They built the model for the scooter behaviour in MATLAB®/Simulink® and computed it with a DS1005 PPC Board. The I/O signals were generated and measured by the DS2210 HIL I/O Board, which also performed the signal conditioning. This board contains special functions for generating and reading ECU crank-angle-based signals with high accuracy and convenience.

Figure 3-12 shows the set up for Piaggio MP3 scooter HIL test with dSPACE tools (dSPACE GmbH 2007).

Figure 3-12 The hardware-in-the-loop setup with a dSPACE Simulator Mid-Size

3.4.1.2.5 Integration test

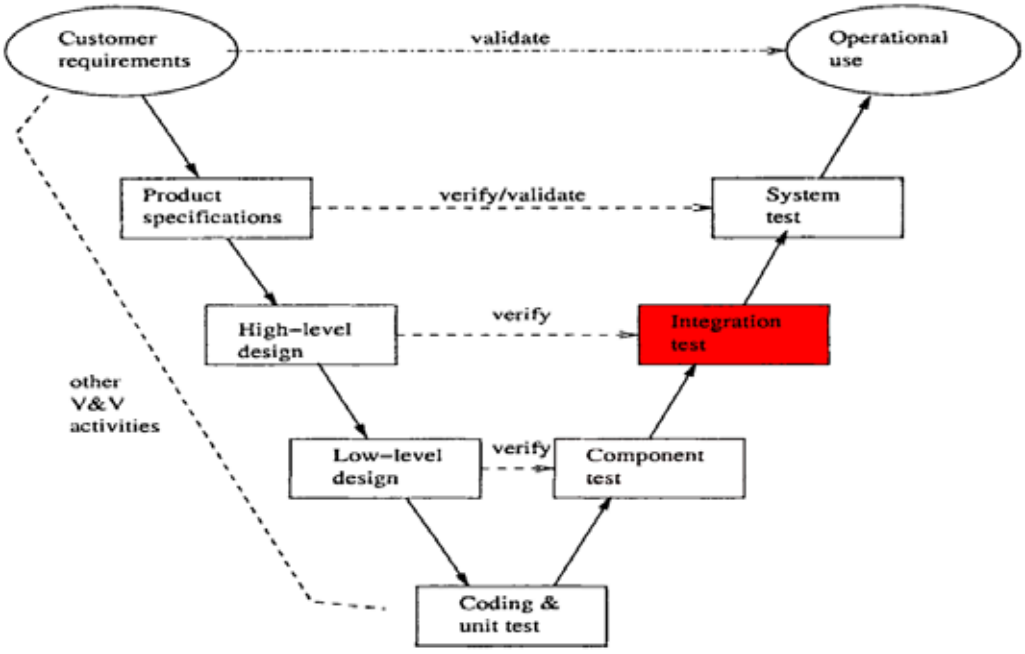


Figure 3-13 V model (Integration test)

The aim of integration testing is to make sure different parts of system are able to correctly work together. (McGregor and Sykes 2001) "Integrated tests are performed to test various parts of the system (components, modules, applications,

etc.) that were separately developed in a set." (Dasso & Funes 2006, p76). The integration test starts after each function is complete and passed its own unit test.

Procedure to implement integrated tests(Dasso & Funes 2006, p77):

1. *Prepare test environment, using test data and test server, which are configured to simulate the production environment.*
2. *Identify test cases based on requirements and architecture.*
3. *Detail procedures for each test case.*
4. *Implement integrated tests.*
5. *Execute integrated tests.*
6. *Analyse results. If errors are found, they must be registered in the problem reports tool and associated with those responsible for the corresponding corrections. If none are found codification may stop.*
7. *Fix problems encountered.*
8. *Execute tests again. After ending it, return to step 6.*

The integration test covers the hardware and the software. Due to the dependency of the different software components and different hardware parts. It is very important and useful to make a strategy to do the integration test. There are three fundamental strategies: Big bang, Bottom-up, and Top-down. They are not mutually exclusive, so a variety of different strategies can be combined. The choice of strategies depends on factors such as availability of the integration parts (e.g. third party software or hardware); size of the system; whether it is a new system or an existing system with added/changed functionality; and the system architecture (Broekman & Notenboom 2003, p46).

Big bang integration

This strategy is quite simple, all modules are integrated together, the whole system is tested. It can be successful if: a large part of the system is stable and

only a few new modules are added; the system is rather small; the modules are tightly coupled and it is too difficult to integrate the different modules stepwise.

Bottom-up integration

This strategy starts with low level modules with the least dependencies, using drivers to test these models. It can be applied to build a system step by step; the subsystem is developed in parallel and then integrated together. The integration can start very early in the development process. The advantage is this strategy can detect interface problems early. The disadvantage is that many drivers are used, and consume lots time because of the iteration of tests.

Figure 3-14 is a module call graph. A bottom up integration is illustrated in Figure 3-15.

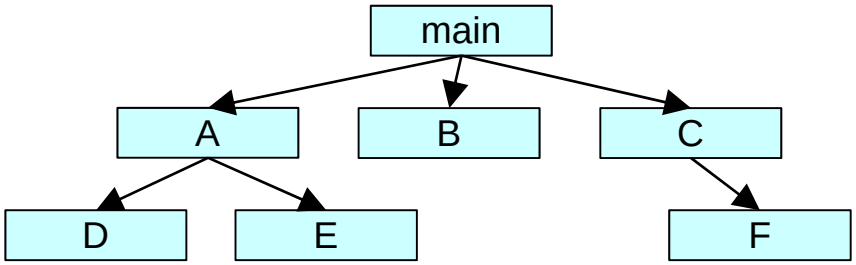


Figure 3-14 module call graph

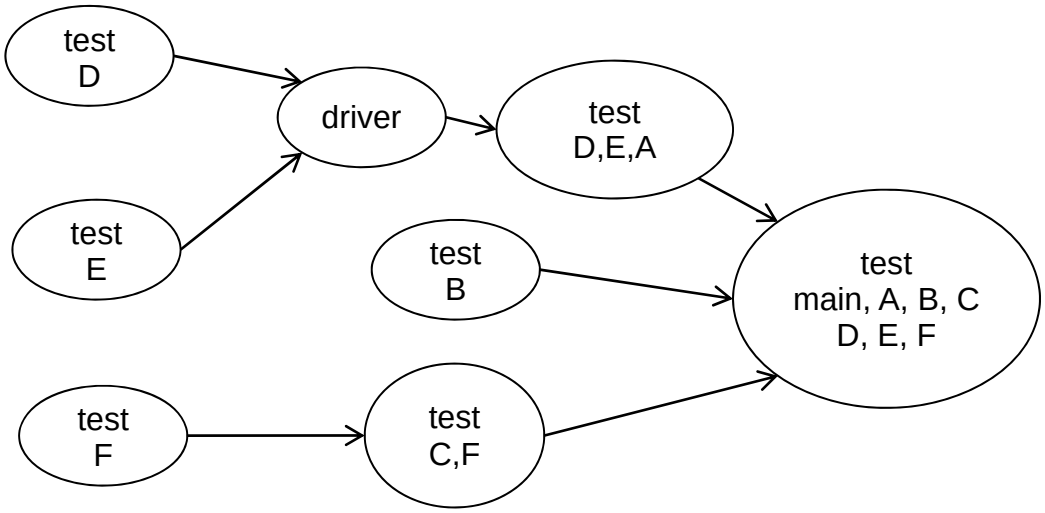


Figure 3-15 bottom-up integration

Top-down integration

This strategy offers the ability to integrate the modules top-down starting with the top-level control module. The top-level module is tested at beginning and the lower level modules progressively added one by one. The lower level modules normally are simulated by stubs. The advantage of top-down integration is that it can give an overview of the entire system. The disadvantage is that a change of requirements, that will affect the low level modules, may lead to changes in top-level modules. A large number of stubs are needed to test every integration step.

For the call graph in Figure 3-14, top-down integration can be illustrated in Figure 3-16.

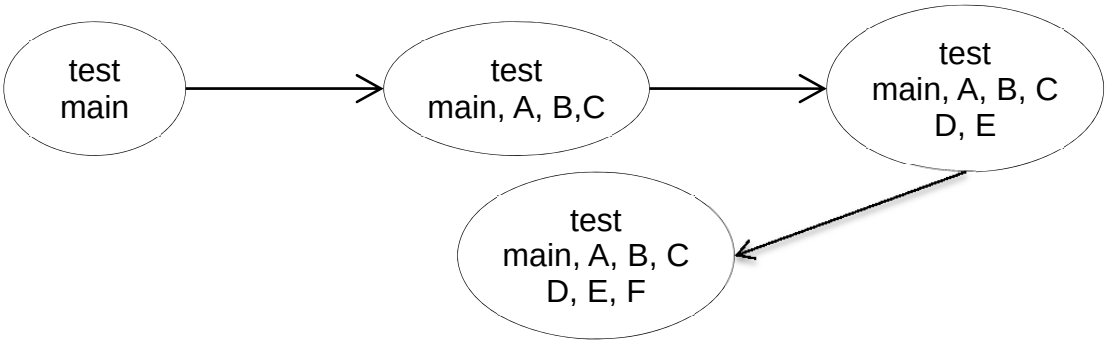


Figure 3-16 top-down integration

In the automotive industry, the integration test involves hardware and software. Often the system works in the simulated model or prototype, but won't work for the hardware integration. Hardware-In-The-Loop can also be used for integration testing. The ECUs may be connected to different networks (e.g. CAN, FlexRay, and LIN etc.). They communicate with each other by message passing. During the message transmission, many things could happen to affect the message arriving at the destination, so we need some mechanism to record the whole system's behaviour which includes local ECUs and network. There are some tools to help to monitor the network of ECUs, e.g. CANoe, CANalyzer etc., but they only record the data on the network and cannot record any synchronised information about the local ECUs. So a method is needed to be able to record whole system state (global state). In a later chapter is decribed more detail about how to build and analyse the global system state.

3.4.2 Validation

Validation is the process confirming that it meets the user's requirements. It checks if the system meeting the users' need.

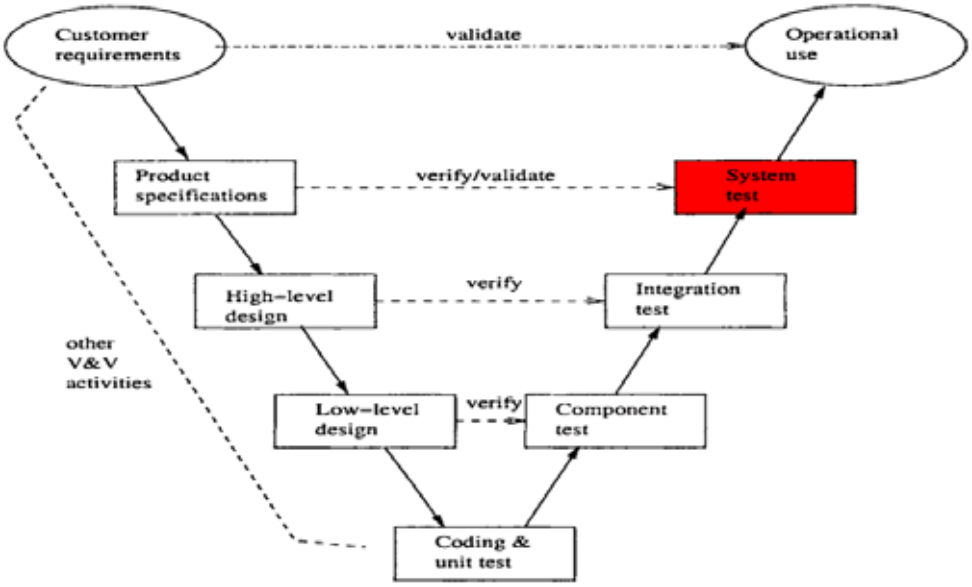


Figure 3-17 V model (system test)

The early system validation will be checked by using some animation techniques: formal specification, modelling and simulation techniques, and using software to simulate the system. These techniques are used in the early design and development phases. Computer algorithms can be used to mathematically prove that the system specifications are consistent.

3.4.2.1 Formal specifications validation

"A formal specification is simply a description of a system using a mathematical notation." (Bowen 2003, p4). Formal specification can clearly explain the system, without any confusion as the mathematics is precise. After the developer acquires the requirements from the user, they can build a system specification with the mathematical notation. This specification is able to describe exactly the system to apply on the requirements. In order to check if the system is satisfies the users' requirements, after the system formal specification is built; the developer needs to meet the users again to show them this specification. This should be the first validation of the system.

There are different mathematical notations. The notation widely used by the computer is the Z notation. Z notation that is has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG) within the Computing Laboratory (Bowen 2003, p4). For example, {n : Z • 2 * n} means a set of all even integer. n is a variable, Z represents integer type, any integer multiplies by 2, the result will be an even integer. This is an easy example about denoting sets by using Z notation. It also can denote logic, types, structure, relations, functions, etc.. The big advantage of using Z notation to represent the requirements of the user reduces the ambiguity, as the nature of the formal specification. It helps the earlier system validation and enhances the validation of the system.

3.4.2.2 Model based development

"Modelling is the process of producing a model; a model is a representation of the construction and working of some system of interest. A simulation of a system is the operation of a model of the system." (Maria 1997) Model is the static "shape/body" of the system. The simulation is the dynamic "activities" of the system model. The model test is known as Model In the Loop (MIL).

A system model is built of the behaviour and the structure of the system. System structure defines the interaction among the components. System behavior defines the how the components change state, it may be caused by the communication among themselves.

Figure 3-18 is an example of the structure of a car speed control system. If the driver presses the accelerate pedal or the button to set a constant speed; the force

sensor that is connected with the pedal or button sends a signal to the speed controller; the speed controller tells fuel controller. How much fuel to feed, and this is shown in the diagram as the “accelerate” relationship. This is the system structure.

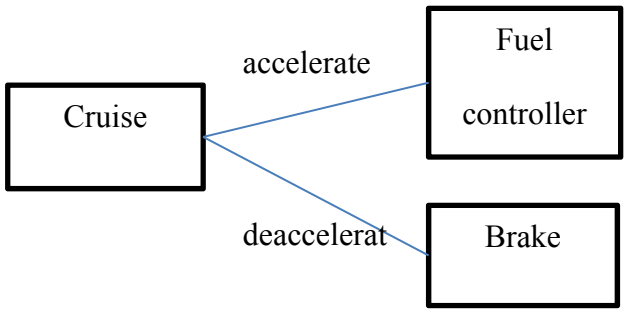


Figure 3-18 Basic structure for an car speed control system

In automotive system design, the system and platform on which system is being implemented (e.g. ECU is a platform in the automotive industry) both need to be considered, because they are developed in parallel . But the platform plays less or no role in the normal software design. This is the main difference between normal software design and automotive software design. The combination of platform model with system model forms one huge unified model. The platform model could offer some guarantees which could be used in the debugging/validation of the system model. The problem is how to relate the platform validation and system model validation? In the car speed control system, system model describes speed controller, fuel controller, and brake controller running on the different ECUs and communicating each other via a network. The platform model describe all the ECUs' connection architecture(processing elements connected via network) and the communication behaviour (in the form of the network protocol through

which the processing elements communicate). The platform model of a car speed control system is described in Figure 3-19

In this case, the guarantees to validate the protocol for system model are:

- the network should be always connected
- The messages in the channel should have the correct priority. (Brake controller should have highest priority safety resons.)
- The message should be passed in a limited/useful time.

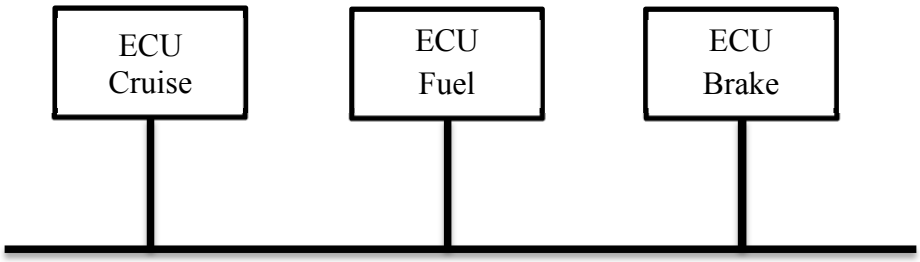


Figure 3-19 Physical diagram for a car speed control system.

Normally the model design depends on the requirements of the user. The requirements are analysed by the developers. The model design should have following properties:(Roychoudhury 2009, p11)

- Complete—The model should be a complete description of system behaviour.
- Based on well-accepted modelling notations/standards.
- Preferably executable—ideally the model is equipped with execution semantics, so that simulations can be run on the model itself.

After building the model, it should be operated to see how it works. Simulating a model can discover some unexpected behaviour by random simulation. The requirements of user are implemented in the model, to find out if the system operates as expected. Even the user can be asked to operate the model and check

out if any problems. It is the main reason why model simulation is used. Model simulation validates if the system model satisfies the requirements of user.

A simulation model builds a link between informal requirements and the system model. Also there should be a link between the model and the real system; this can be done by code generator. The source code of system can be generated by the model compiler and compiled by the compiler toolset. After the source code is compiled, the binary code is generated. The binary code is flashed into the hardware. The hardware can be tested against the model, see if the hardware implementation matches the model. This process is called model based system development. It is illustrated in Figure 3-20 (Roychoudhury 2009, p51)

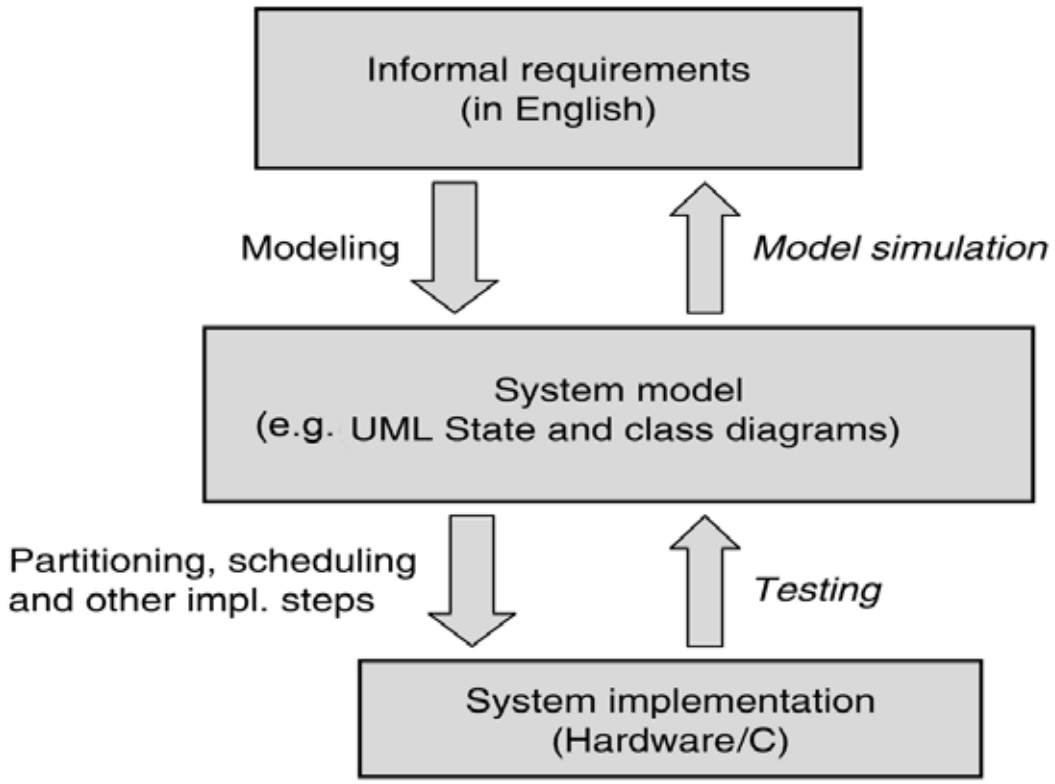


Figure 3-20 model based system development

In automotive software development, math works Simulink is commonly used for model based development. Simulink is an important component of the MATLAB.

It offers dynamic system modelling, simulation, and comprehensive analysis integration environment.

Simulink offers all mathematically simulated models. Each Simulink model is a block diagram. After the system is modelled by the block diagram, the model can be run on Simulink. Therefore it gives the initial overview of the system execution situation. Figure 3-21shows Simulink library browser.

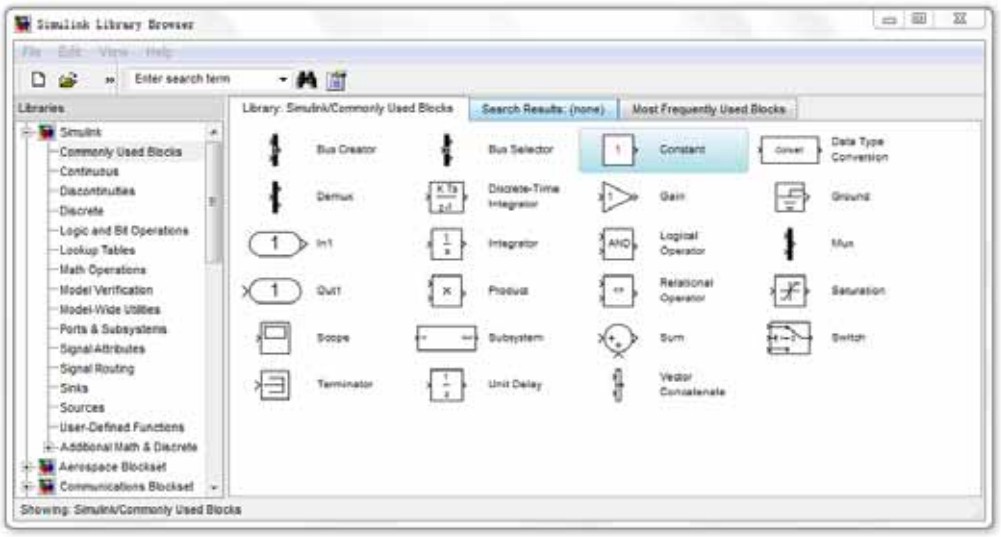


Figure 3-21 Simulink library browser

Based on Simulink, dSPACE developed TargetLink that is integrated with Simulink. The following example is a company called “Delphi Electronics & Safety” using dSPACE tools to develop new algorithms for power window functions (dSPACE GmbH 2010).

> “The entire functionality of the power window control was designed in Simulink/TargetLink and then autocoded with TargetLink. The generated code was highly efficient and clearly structured. Moreover, simulation in MIL and SIL modes proved extremely useful in advancing controller design and fixed-point software development. For offline simulation,

signals recorded in rapid control prototyping were reused, and additional test vectors were also developed (Figure 3-22). To specify the position control's software interfaces, the TargetLink Property Manager was used frequently to convert the Stateflow sections of the anti-pinch protection into production-ready C code. TargetLink's ability to flexibly generate code for look-up tables was harnessed to autocode the stall detection, making it possible to use different types of search and interpolation routines, to partition the code into different files, etc." (dSPACE GmbH 2010)

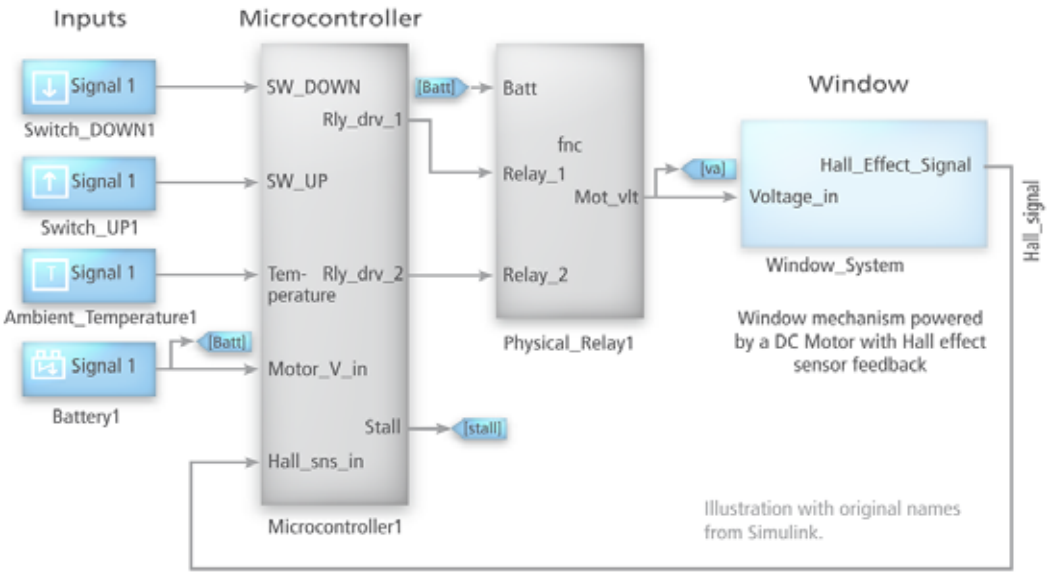


Figure 3-22 Simulation environment in Simulink/TargetLink.

3.4.2.3 Rapid prototype validation

The prototype embedded system is tested while connected to the real environment. This is the ultimate way of assessing the validity of the simulation model. Figure 3-23 shows the process for the rapid prototype development.

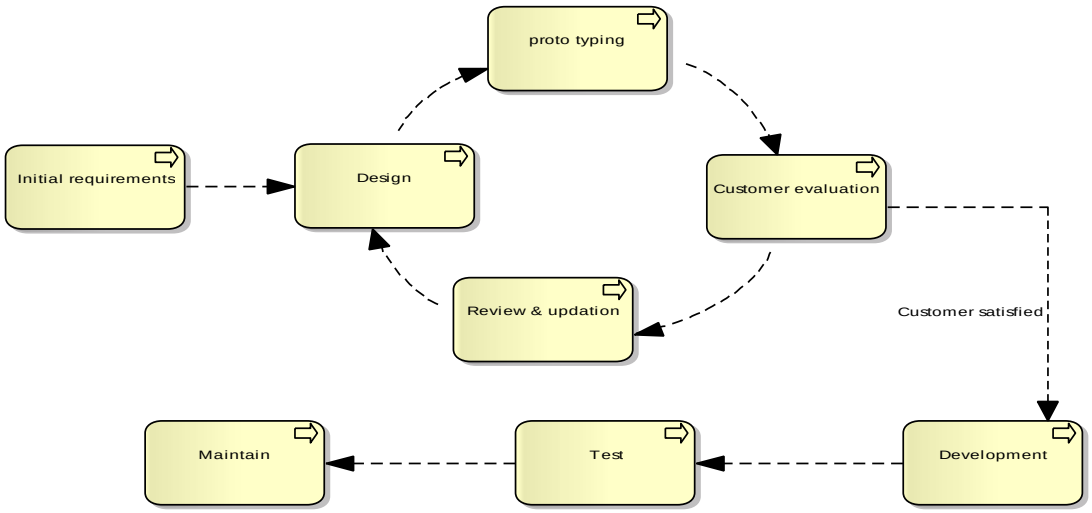


Figure 3-23 rapid prototype development

The system test and acceptance test are used to validate if the system satisfies the requirements of the user by installing the software in the real hardware. Performance testing can be carried out which focuses on the performance of the software system in realistic operational environments. Many such systems are real-time systems, where timely completion of computational tasks and overall workload handling are of critical importance (Tian 2005, p213).

Stress testing, which is a special form of performance testing can also be done, where software system performance under stress is tested. This type of testing is also closely related to capacity testing, where the maximal system capacity is assessed (Tian 2005, p213).

The prototype validation also involves system parameters evaluation. In the automotive industry there are some tools developed for the rapid prototyping hardware; CANape is a tool that is developed by Vector, it is available for ECU development, calibration, and diagnostics as well as for measurement data acquisition.

The primary application area of CANape is in optimizing parameterization (calibration) of electronic control units. It can calibrate parameter values and simultaneously acquire measurement signals during system runtime. The physical interface between CANape and the ECU might be made via the CAN bus with CCP, for example, or via FlexRay with XCP. Additionally, with its Diagnostic Feature Set CANape offers symbolic access to diagnostic data and services. As a result it has all relevant integrated functions for measurement, calibration, flashing and diagnostics. Its reliance on standards makes CANape an open and flexible platform for all phases of ECU development. Figure 3-24 shows CANape measurement configuration window.

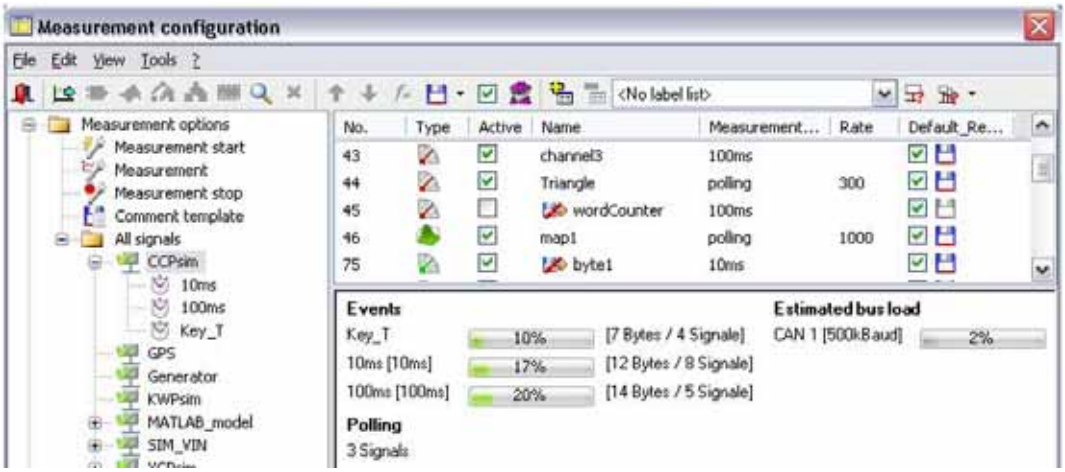


Figure 3-24 CANape measurement configuration

3.5 Automotive distributed system integration

There are some companies that develop the tools for the automotive distributed system integration test. The most popular two companies are Vector and dSPACE. Vector offers the CANoe, CANalyzer, CANdela, CANape, etc. for the network analysis, ECU diagnostic and calibration. CANoe is a very powerful development tool for the distributed automotive system; it offers the service to do the ECUs

integration and the restbus simulation is supported. It connects all or part of ECUs (simulated or/and real) together to implement the integration test. During the test CANoe logs all communication messages to the CANoe log file. To use CCP (see section 2.6), the local state of each node also can be logged into the CANoe log file. Therefor the CANoe log file records all communication messages and the local states of each node. However from this log file, the variable relationship between the nodes cannot be clearly observed, e.g. if the node A send message M, how M is going to affect the other nodes or have no affect at all because M does not satisfy the threshold of the nodes, or what happens if M delayed. The message delay, halt and modification (bits lost during the transmitting) is the major problem for the distributed systems. Some of them are very easy to be observed, e.g. termination: the whole system just shut down, deadlock: all nodes are in the waiting state. However, failure of distributed systems is not easy to be observed (unpredictable), because of the arbitrary of the distributed system. Every time starting the distributed system, it may go through different execution states (global states). In the automotive system, some of global states may reduce the safety of the vehicle, increases the other risks and be uncomfortable for the customers etc., so to find fault global states is essential for the ECU integration. The global states of the system are more about the global view of all local states of all ECUs. CANoe does not have a function to construct the global states of the system; however, it offers potential abilities to construct the global states. The next section is going to talk about how the software integration testing is done by the industry.

3.5.1 Automotive Software integration testing

The automotive software integration test can be done by two approaches: top-down integration or bottom-up integration. Both approaches are supported by the

most popular automotive development tool companies Vector and dSPACE. The CANoe tool of Vector can run the whole system in which all nodes are simulated as soon as whole system is specified or partly specified. Each time after the real ECU is developed, replace the simulated node with the real node on the restbus simulation. The system will be integrated progressively by replacing the ECUs one by one. The top-down integration of CANoe is shown in Figure 3-25(Vector Informatik GmbH 2010b, p23).

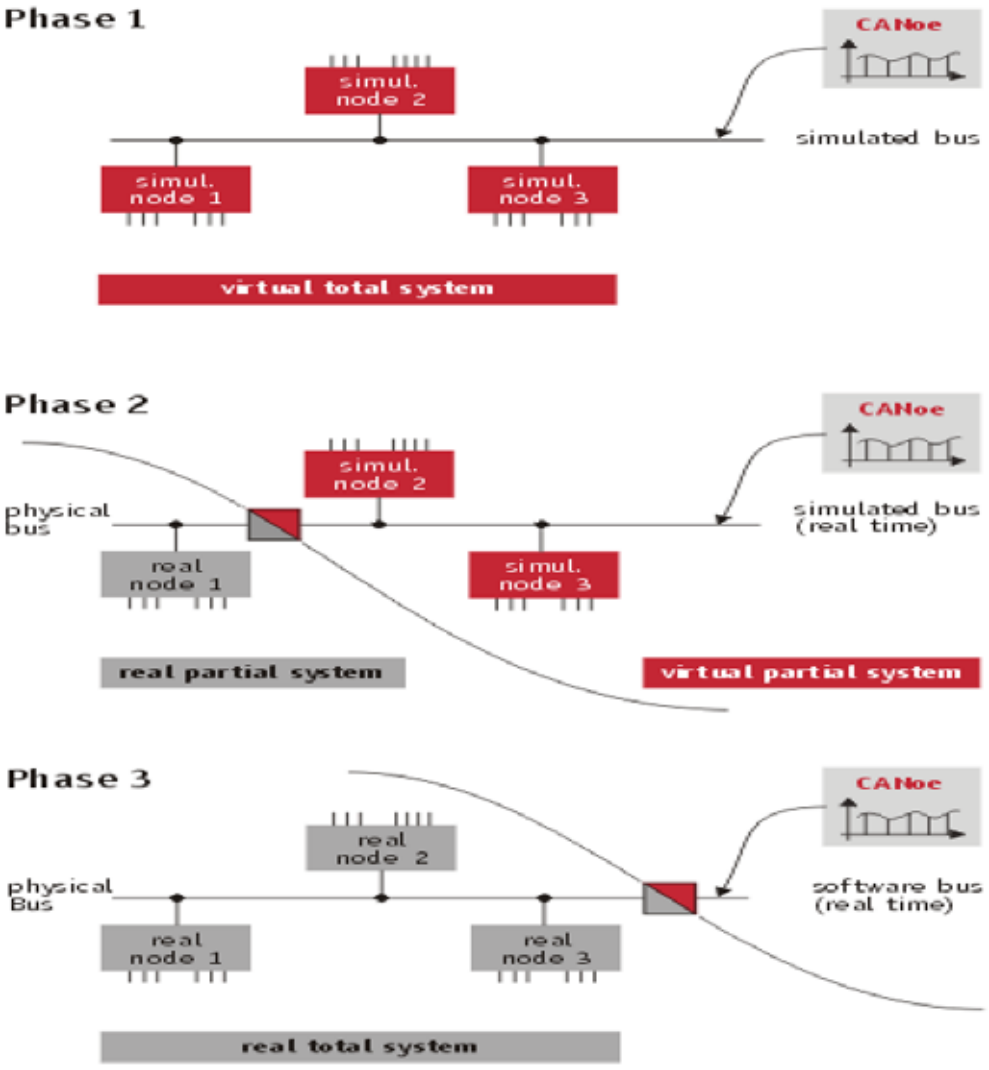


Figure 3-25 CANoe top-down integration

Phase 1: Requirements analysis and design of the network system.
Phase 2: Implementation of components with simulation of remainder of the bus
Phase 3: Integration of the overall system

Also the bottom up approach can be done by using CANoe. Every time when the least dependency ECU is developed, integrating the ECU to the CANoe, the system can be integrated one by one.

dSPACE uses the HIL simulation to do the system integration testing (dSPACE GmbH 2009, p102). They also offer the restbus simulation as well as CANoe. The difference is CANoe using program code to simulate the ECU nodes, the dSPACE simulation using the hardware simulator to simulate the ECU. The dSPACE simulator is illustrated in Figure 3-26.

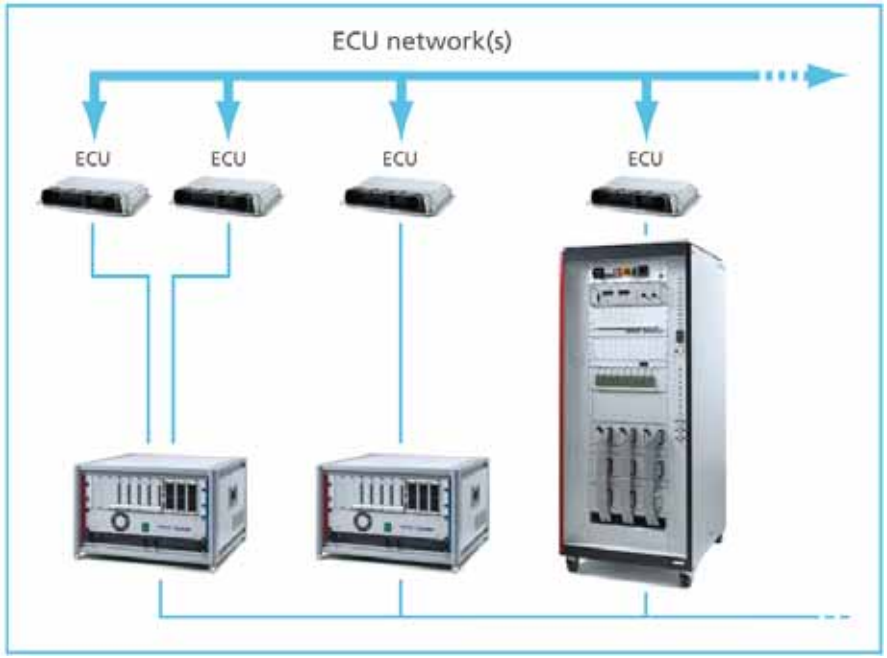


Figure 3-26 dSPACE simulator

3.5.2 Integrating time-triggered system

As discussed in section 2.5.2, the timer-triggered system has the global clock that synchronized the system, the global states can be easily constructed or the system can be paused in the same global time. It makes the integration of the time-triggered system is much easier.

3.5.3 Integrating event-triggered system

Because the event-triggered system does not have global clock as time-triggered system has or the shared memory, it makes the integration test much harder than the time-triggered system. The system execution global states trace is unpredictable; it may change on the different running. The unexpected global states are very hard to be detected by the current development tools. There are no such tools in the automotive industry to construct global execution states for the ECU network.

3.6 Conclusions

Testing is very important during software development, also for the automotive software; it goes through entire development life cycle. This chapter described the V model that is the fundamental development methodology, and the variants of the V model (multiple V and nested V). The test plan states how to apply the different tests to the corresponding development phase in the V model.

In general there are two types of faults in the software product; specification and implementation fault; specification faults fail the requirement of the user, it does not do the thing that user expect properly, so developers need to validate the specification that is delivered from requirements of the user. Implementation fault fails the specification that has been written, it does not correctly implement the function that has been specified, so developers needs to verify if the function satisfies the specification. The validation of a system requires a global overview of the system to check if it works as it expected. Verification checks the detailed implementation of the system. The different test types are applied to check these two fault types. A number of common automotive system integration problems were discussed as well as the difficulties in detecting them.

References

A.Al-Ashaab, S.Howell, A.Gorka, K.Usowicz, & P.Hernando Anta Set-Based Concurrent Engineering Model for Automotive Electronic/Software Systems Development, *In CIRP Design Conference 2009*, p. 464.

Bowen, J. 2003. *Formal Specification and Documentation using Z: A Case Study Approach* Thomson Publishing.

Broekman, B. & Notenboom, E. 2003. *Testing Embedded Software*.

Dasso, A. & Funes, A. 2006. *Verification, Validation and Testing in Software Engineering* Idea Group Publishing.

dSPACE GmbH. HIL for a Three-Wheeler Scooter. 2007. dSPACE .

dSPACE GmbH. dSPACE Catalog 2009. 2009. dSPACE.

dSPACE GmbH. Making Power Windows safe. 2010. dSPACE GmbH, Paderborn, Germany, dSPACE Magazine.

dSPACE, G. dSPACE Catalog 2009. 2009. dSPACE,GmbH.

Everett, G.D. & McLeod, R. 2007. *Software Testing Testing Across the Entire Software Development Life Cycle* Wiley-IEEE Computer Society Press.

Maria, A. Introduction To Modeling And Simulation. 1997. ACM.

McGregor, J.D. & Sykes, D.A. 2001. *A practical guide to testing object-oriented software* Addison-Wesley.

Michael Schneider, Johnny Martin, & W.T.Tsai. An Experimental Study of Fault Detection In User Requirements Documents. 1, 188-204. 1992. ACM Transactions on Software Engineering and Methodology.

National ITS Architecture Team 2007, *System Engineering for Intelligent Transportation Systems*.

Patton, R. 2005. *Software Testing* Sams Publishing.

Pressman, R.S. 2001. *Software Engineering*, 5th ed. *Thomas Casson*.

Roychoudhury, A. 2009. *Embedded Systems and Software Validation* Morgan Kaufmann.

Schaeuffele, J. & Zurawka, T. 2005. *Automotive Software Engineering Principles Processes Methods and Tools* SAE International.

Tian, J. 2005. *Software Quality Engineering Testing, Quality Assurance, and Quantifiable Improvement* John Wiley & Sons, Inc., Hoboken, New Jersey.

Vector Informatik GmbH. Product Catalog. Development of Distributed Systems ECU Testing. 2010a. Vector Informatik GmbH.

Vector Informatik GmbH. User Manual CANoe Version 7.5. 2010b.

Chapter 4 Logical Time

4.1 Introduction

Time is important to determine the order or causality of events. The clock is the mechanism used to record time. The causality (order) of events is a very important issue in distributed system and is a fundamental concept for analysing and debugging the system state. Physical time is used to track the causality of events using synchronized clocks. However it is impossible to apply a global physical time in distributed system made up of many separate CPUs devices. This is because the devices, which can be either the same or different types, may have different CPU clock frequencies, the accuracies of which can vary with temperature, moisture and manufacturing tolerances. Accurate clock synchronization is therefore impossible to achieve over a period (Nicola Santoro 2007, p333).

For example Figure 4-1 shows 4 nodes, all connected by a bus line. Each node has its own local time above it. If node2 sends message to node1 at local time 3:55 and node1 receives the message at local time 3:35, it looks very confusing that the receiving message event happens before the sending event. Such unsynchronized time is not good for distributed system analysis or testing; so a common time (Global time) is needed for the purpose of distributed system analysis and testing.

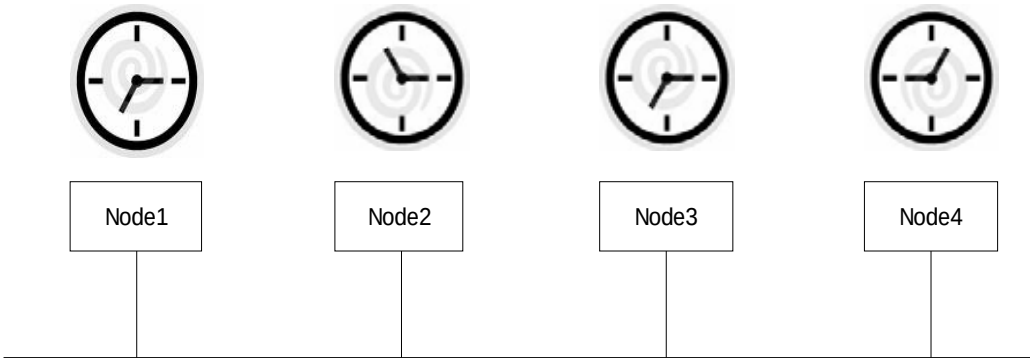


Figure 4-1 Four node distributed system with physical clocks

Although there can be no common synchronized global physical time in a distributed system there are some mechanisms (clocks) that can help us to approximately realize global physical time. This chapter introduces these mechanisms and describes their principles of operation.

4.2 Logical time

In general a logical clock can measure the causalities of events. In a distributed system, every node has its own local clock that is advanced using a set of rules. Each event is assigned a timestamp and the causalities of events can be inferred from the timestamps. The timestamp follows the basic monotonic property; if an event *a* causally effects event *b*, then the timestamp of event *a* should be smaller than the timestamp of event *b*.

A system of logical clocks consists of a time domain T and a logical clock C. Elements in T are an ordered set over a relation <. This relationship can be called "happens before" or causal precedence (denoted by →) and is also the same as the "earlier than" relationship provided by physical time. The logical clock is a

function that maps the event e in a distributed system to an element, denoted as C(e) and called the timestamp of e in the domain T. The clock is defined as

$$C: H \mapsto T$$

To satisfy the property

$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$$

This monotonic character is called the clock consistency condition.

If for event e_1 and e_2

$$e_1 \rightarrow e_2 \equiv C(e_1) < C(e_2)$$

then the system of clocks is strongly consistent. (Ranal and Singhal 1996)

There are two requirements for implementing logical clocks

1. A data structure for the local process to represent its logical time;

2. A protocol (set of rules) to update the data structure and ensure the protocol follows the consistency condition.

A data structure for each process has two functions:

- As a local logical clock, denoted by lc_i, that helps process p_i measure its own progress.
- As a logical global clock, denoted by gc_i, that is a representation of process p_i's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, lc_i is a part of gc_i.

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

- **R1** - This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
- **R2** - This rule governs how a process modifies its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Different systems may use different data structures to represent the logical time and different protocols to update the data structure. However, all logical clock systems should implement R1 and R2 and consequently ensure the fundamental monotonicity property associated with causality. Moreover, the different logical clock systems may provide some additional properties to its user.

4.3 Scalar time

Scalar time was proposed by Lamport (Leslie Lamport 1978) and attempts to totally order events in the distributed system. In the scalar time system, time is represented by positive natural numbers. The logical local clock of a process P and its local view of the global time are expressed by one integer variable C.

Rules R1 and R2 to update the clocks are as follows:

- **R1** - Before executing an event (send, receive, or internal), process P_i executes the following:

$$C_i := C_i + d \ (d>0)$$

In general, every time R1 is executed, d can have a different value, and this value may be application-dependent. However, typically d is kept at 1 because this is

able to identify the time of each event uniquely at a process while keeping the rate of increase of d to its lowest level.

- **R2** - Each message piggybacks the clock value of its sender at sending time. When a process P_i receives a message with timestamp C_{msg}, it executes the following actions:

  1. $C_i := \max(C_i, C_{msg})$;
  2. execute R1;
  3. deliver the message.

The sequencing of scalar clocks between three communicating processes is illustrated in

Figure 4-2. The events of each process are shown on horizontal timelines with their local scalar timestamps. Messages transmission events between processes are shown using arrows.

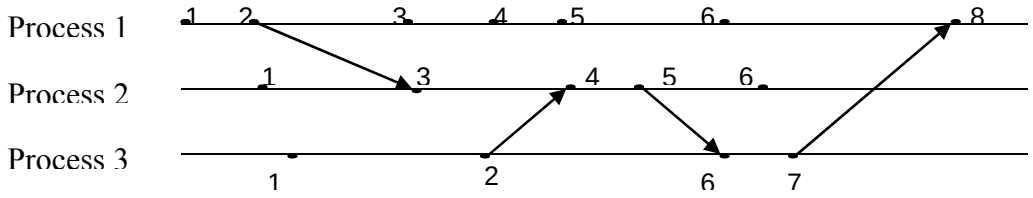


Figure 4-2 scalar time

Gap Detection:

Sometimes because of the delay of the message transmission the receiving message timestamp may be smaller than the current scalar time of the receiving

process. This is because scalar time lack of the overview of the global time, what is called the gap detection property. The gap detection can be used to detect the message delay or the message undelivered; they are the main problems that could happen on the distributed system, so the gap detection is a very important issue for the distributed system.

Gap Detection(Ozalp Babaoglu and KeithMarzullo 1993):

> *Given two events e_1 and e_2 along with their clock values $C(e_1)$ and $C(e_2)$ where $C(e_1)<C(e_2)$, determine whether some other event e_3 exits such that $C(e_1)<C(e_3)<C(e_2)$.*

To account for the lack of logical time gap detection, the system needs to set an allowable delay time; if the received message has a timestamp smaller than the allowable delay time, we can call this received message stable. Stable means: A message m received by process p is stable if no future messages with timestamps smaller than TS(m) can be received by p.(Ozalp Babaoglu & KeithMarzullo 1993)

Properties of Scalar Clocks:

Scalar clocks exhibit the following properties.

Consistency property:

Scalar clocks are monotonic and therefore consistent.

For events e_1 and e_2 $\quad e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$

No strong consistency:

Although the scalar clocks are consistent, they are not strongly consistent which is for two events e_1 and e_2 means that if $C(e_1)<C(e_2)$ then this does not necessarily mean that $e_1 \rightarrow e_2$.

Total ordering:

All timed system events can be put in an ordered set; the whole set being ordered by the timestamps. However, the problem is that there may be two or more events in the different processes that have the same timestamps. The notation < expresses any arbitrary ordering of the processes. If event e_i occurs in process p_i and event e_j occurs in process p_j, then $e_i \Rightarrow e_j$ if and only if

(i) either $C(e_i) < C(e_j)$ or

(ii) $C(e_i) = C(e_j)$ and $p_i < p_j$.

Because events can occur at the same logical scalar time independently, they can be ordered in any arbitrary criterion without violating the causality relation. A total order is generally used to ensure liveness properties in distributed system. Liveness means a message that arrives at a process must eventually be delivered to the process.

Event counting:

If the timestamp of events always increases by a known number (normally 1) and event e has a timestamp t_e, then the minimum event logical duration can be calculated by t_e-1. This is the number of events handled by the process before event e occurs.

4.4 Vector Time

In the previous section, the causal history of events in distributed system was measured by a scalar time, which is a local view of the global time condensed into one integer variable. Scalar time combines the global time and local time together,

using one non-negative integer to measure global and local time; it lacks the global view of the system.

Alternatively, a fixed-dimensional vector can represent the causal history. The mechanism of vector clocks was developed individually by Fidge (Colin Fidge 1991) and Mattern (Friedemann Mattern 1998). Vector time assigns the times of processes in a vector; each process has a vector time which contains other processes' local time, so every process the global view of the whole system.

The implementation of Vector time is as follows:

Each process p_i maintains a vector time VT_i. If the total number of processes is n, the vector time of a process can be expressed as $VT_i[1..n]$ where

(i) The element $VT_i[i]$ is local clock of the process p_i;

(ii) $VT_i[j]$ ($j \in x:N \mid x \in 0..n \ \wedge x \neq i \}$) is the latest knowledge that P_i has of the local clock of process P_j.

Process p_i uses the following two rules R1 and R2 to update its clock:

R1 - Before executing an event, process p_i updates its local logical time as follows:

$$VT_i[i] := VT_i[i] + d \qquad (d > 0, \text{normally is } 1).$$

R2 - Each message m is piggybacked with the vector clock VT of the sender process at sending time. On the receipt of such a message (m,VT), process p_i executes the following sequence of actions:

- update its global logical time as follows:

  $1 \leq k \leq n : VT_i[k] := \max(VT_i[k], VT[k]);$

- execute R1;
- deliver the message m.

Figure 4-3 shows an example of vector time.

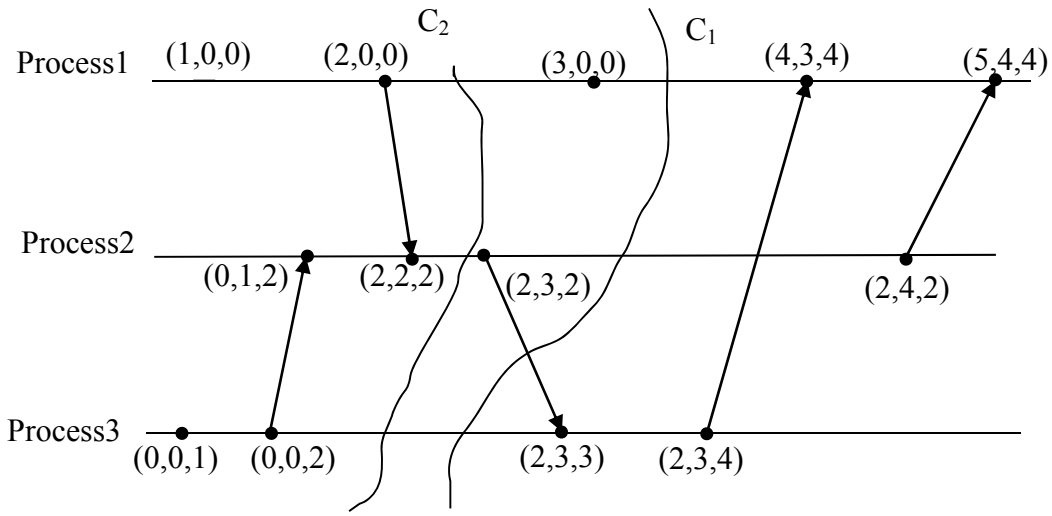


Figure 4-3 vector time

Properties of Vector Time

This section describes the useful properties of Vector clocks and illustrates practical uses of these properties.

Given two n-dimensional vectors V and V' of natural numbers, we define the "less than" relation (written as <) between them as follows

$$V < V' = (V \neq V') \wedge (\forall\ k{:}1 \leq k \leq\ n{:}\ V[k] \leq V'[k])$$

Property 1 Strongly Consistent Clock Condition

If event e happens before event e' then vector time of e is smaller than vector time of e'. Another way of putting this is if the vector time of e is smaller than vector time of e' then e happened before e'.

$$e \rightarrow e' \equiv VT(e) < VT(e').$$

It is not necessary to know on which processes the two events were executed. If this information is available, causal precedence between two events can be verified through a single scalar comparison.

Property 2 Simple Strong Clock Condition

Given event e_i of process p_i and event e_j of process p_j, where $i \neq j$

$$e_i \rightarrow e_j \equiv VT(e_i)[i] \leq VT(e_j)[i]$$

$VT(e_i)[i] = VT(e_j)[i]$ represents that e_i is the latest event of p_i which causally precedes e_j of p_j, so e_i must be a send event.

Property 3 Concurrency

Given event e_i of process p_i and event e_j of process p_j

$$e_i \parallel e_j \equiv (VT(e_i)[i]>VT(e_j)[i]) \wedge (VT(e_j)[j]>VT(e_i)[j])$$

the two events in different processes happen at the same time.

In the example process1 event with vector time (4,3,4) and process2 event with vector time (2,4,2) are concurrent events.

Property 4 Pairwise Inconsistent

Event e_i of process p_i is pairwise inconsistent with event e_j of process p_j, where $i \neq j$, if an only if

$$(VT(e_i)[i]<VT(e_j)[i]) \vee (VT(e_j)[j]<VT(e_i)[j])$$

"A cut is a line joining an arbitrary point on each process line that slices the space–time diagram into a PAST and a FUTURE." In Figure 4-3, cut C_1 is a pairwise inconsistent. In this situation happens, the cut includes at least one receive event without the corresponding send events.

A consistent cut contains no pairwise inconsistent events. In vector clock terms, the property becomes

Property 5 Consistent Cut

A cut defined by (c1;. . . cn)is consistent if and only if

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VT(e_i^{ci})[i] \geq VT(e_j^{cj})[i]$$

This expression means if the cut is consistent, then all events in the cut can track back to the previous events which precede current event without losing any send event. In Figure 4-3 the cut C_2 is a consistent cut.

Property 6 Counting

Given event e_i of process p_i and its vector clock value $VT(e_i)$, the number of events e such that $e \rightarrow e_i$ (equivalently, $VT(e) < VT(e_i)$) is given by $\#(e_i)$.

Property 7 Weak Gap Detection

Given Event e_i of process p_i is pairwise inconsistent with event e_j of process p_j, if $VT(e_i)[k] < VT(e_j)[k]$ for some $k \neq j$, then there exists an event e_k such that

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Three random events e_i, e_j and e_k, so it cannot be concluded these events forms a causal chain $e_i \rightarrow e_j \rightarrow e_k$.

4.5 Matrix Time

The logical time can be represented by a set of n×n matrices of non-negative integers. A process p_i maintains a matrix $mt_i[1..n, 1..n]$ where,

- $mt_i[i,i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i;

- $mt_i[i,j]$ denotes the latest knowledge that process p_i has about the local logical clock, $mt_j[j,j]$, of process p_j (note that row, $mt_i[i,.]$ is nothing but the vector clock $vt_i[.]$ and exhibits all the properties of vector clocks);
- $mt_i[j,k]$ represents the knowledge that process p_i has about the latest knowledge that p_j has about the local logical clock, $mt_k[k,k]$, of p_k.

The matrix clock of mt_i contains the local view of the global time. The matrix timestamp of an event is the value of the matrix clock of the process when the event is executed.

Process p_i uses the following rules R1 and R2 to update its clock:

- R1: Before executing an event, process p_i updates its local logical time as follows:

$mt_i[i,i]:=mt_i[i,i]+d$ $\quad$ ($d>0$, normally d is 1).

- R2: Each message m is piggybacked with matrix time mt. When p_i receives such a message (m,mt) from a process p_j , p_i executes the following sequence of actions:
  1. update its global logical time as follows:
     a) $1 \leq k \leq n$: $mt_i[i,k]:=\max(mt_i[i,k], mt[j,k])$, (that is, update its row $mt_i[i,*]$ with p_j's row in the received timestamp, mt);
     b) $1 \leq k, l \leq n$: $mt_i[k,l]:=\max(mt_i[k,l], mt[k,l])$;
  2. execute R1;
  3. deliver message m.

An example of matrix clocks is given in Figure 4-4.

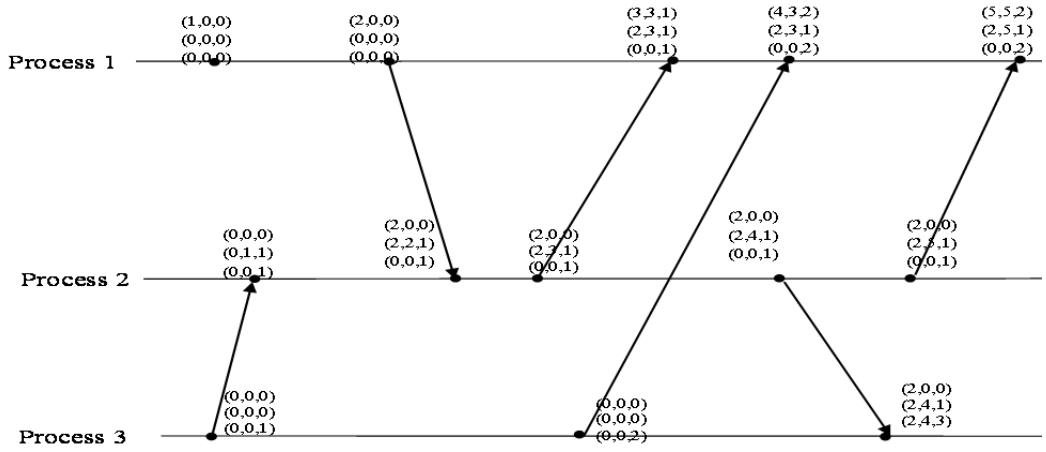


Figure 4-4 Matrix time example

A system of matrix clocks was first informally proposed by Michael and Fischer (M.J.Fischer and A.Michael 1982) and has been used by Wuu and Bernstein (M.J.Fischer & A.Michael 1982) and by Sarin and Lynch (G.T.J.Wuu and A.J.Bernstein 1984) to discard obsolete information in replicated databases.

4.6 Conclusions

Logical Time Mechanism	Storage	Complexity	Global Over View
Scalar Time	Only use a positive integer for each process	Easy to update	None
Vector Time	Each process stores one dimension vector clock that contains Θ(n) components.	Harder	Good
Matrix Time	Each process stores two dimension vector clock, each dimension vector contains $\Theta(n^2)$ comports.	Hardest	Very Good

Table 4-1 clock system comparison

n is the total number of process on the network.

Each one of these logical time measurements have their advantages and disadvantages. Scalar time consumes less resource to process the logical time, but the lack of the global view of the other processes. Vector time gives more detail about other processes, but consumes more resources. Matrix time includes current global time and past global time, but it can consume extremely large resources depends on how many process in the distributed system. Table 4-1 compares the clock systems in their complexity, storage, and global view.

For automotive system testing, to find the relationship of the causality of events in the system can be a good help. All Electronic Control Units (ECUs) in the automotive system are distributed on a network and communicate with each other by message passing. There is no global clock for them to synchronize their local clock; the only way is to use the logical clock to synchronize all the ECUs. By using the global logical clock, the events causality can be defined, and can help to track the execution path of the system.

It seems the vector time is more suitable for the automotive integration because it is more descriptive than scalar time and more efficient than matrix time.

References

Colin Fidge. Logical time in distributed computing systems. 28-33. 1991. *IEEE Computer*.

Friedemann Mattern. Virtual time and global states of distributed systems. 215-226. 1998. *Proceedings of the Parallel and Distributed Algorithms Conference*.

G.T.J.Wuu & A.J.Bernstein. Efficient solutions to the replicated log and dictionary problems. 233-242. 1984. *Proceedings of 3rd ACM Symposium on PODC*.

Leslie Lamport. Time clocks and the ordering of events in a distributed system. 558-564. 1978. Communications of the ACM.

M.J.Fischer & A.Michael. Sacrifying serializability to attain hight availability of data in an unreliable network. 70-75. 1982. *Proceedings of the ACM Symposium on Principles of Database Systems*.

Nicola Santoro. Design And Analysis Of Distributed Algorithms. 2007. John Wiley & Sons, Inc., Hoboken, New Jersey.

Ozalp Babaoglu & KeithMarzullo . Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. 1993. Italy, Laboratory for computer science university of bologna.

Ranal, M. & Singhal, M. Logical Time: Capturing Causality in Distributed Systems. 1996. IEEE Computer.

Chapter 5 Global State and Snapshot

5.1 Introduction

Global state is very useful for system analysis, testing or verifying properties associated with distributed executions. For a single machine, the global state is its own local state; but for a distributed system, each node has running its own process. These processes do not have shared memory, they communicate with each other asynchronously by sending messages. Each component of the distributed system has its own state; the state of a node is defined by its local memory and the active history. The state of network is defined by the sets of messages that pass on the communication channel. The global state is a collection of the local states and the network state.

Recording the global state of a distributed system is a very important issue and can be used in distributed system design in some aspects for example, detecting the stability of a system, deadlocks (Rahul Garg et al. 1994) and termination (K.M.Chandy and L.Lamport 1985), using the global state. In system recovery, each global state can be used as a failure recovery point, just as in windows XP the user can set the recovery point; if the system crashed, it can go back the recover point, or in a database if the transaction crashed, the whole system can roll back to the state before the transaction. For testing the distributed system, the global states can be recorded with a time stamp (logical time normally), ordering them by the time stamp. The execution trace of the distributed system can be built up and the tester can analyse the execution trace to find error.

The snapshot algorithm is used to record the global state. It is very important to have efficient ways to records the global state for the distributed system; but unfortunately there is no shared memory or global physical time.

If there were a shared memory in the distributed system, all processes would have the same view of the data and the entire system states could be considered as one state. If global time were available (each node has the same local clock), the snapshot could record the global state at the same time, and the order of the global state would be absolutely consistent. However, since different node may use the different type of microprocessors, it is impossible to synchronise the clocks of each microprocessor. Even using the same microprocessor the clock cannot be synchronised as it can be affected by the environment (temperature, humidity etc.) and power voltage, particularly for the microprocessors that are used in the automotive industry.

However the shared memory or global physical time is not available for the distributed system. Some snapshot algorithms are developed to help record the global state. Some of them use more memory, some use more channel capacity, the snapshot algorithms have a trade-off between the memory and channel capacity, it mostly depends on the hardware. As there are different communication modes, such as FIFO (First In First Out) and non-FIFO communication channels, the different snapshot algorithm is applied. This chapter introduces some of the snapshot algorithms.

5.2 Snapshot algorithm for FIFO

Originally K. Mani Chandy and Leslie Lamport developed the first snapshot algorithm called Chandy-Lamport algorithm, since then other snapshot algorithms have been derived from Chandy-Lamport algorithm.

5.2.1 Chandy-Lamport algorithm

Chandy-Lamport algorithm (K.M.Chandy & L.Lamport 1985) uses a control message, called a marker. It is used as a state save request signature. The algorithm is only suitable for FIFO systems. There are two rules for processes to implement the algorithm, the marker sending rule and the receiving rule.

Marker sending rule for process p_i

- process p_i records its own state.
- p_i broadcast marker to all process that are connected with p_i .

Marker receiving rule for process p_i

If p_i has not saved its state then execute the "marker sending rule".

Else

save the channel state between the last time state saved to the time p_i received the marker.

(The channel state can be the message send or receive between the time of p_i sending and receiving the marker.)

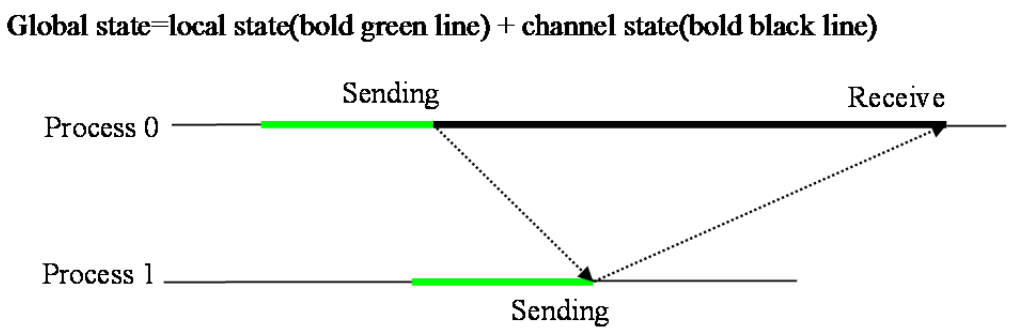


Figure 5-1 Chandy-Lamport algorithm

After the Chandy-Lamport algorithm was developed, some other algorithms were developed variants from it. For example, the Spezialetti-Kearns algorithm (Madalene Spezialetti and Phil Kearns 1986) collects snapshot of concurrent initiation and efficiently distributes the recorded snapshot.

5.2.2 Spezialetti–Kearns algorithm

In the Spezialetti-Kearns algorithm (Madalene Spezialetti & Phil Kearns 1986), each node has unique colour(id_colour) that identifies the node, and local colour(local_colour) that represents the current colour of the node in the particular snapshot.

At the beginning, the local colour for every node is initiated as white. If a initiator node records its local snapshot, then change its local colour to its id_colour, and broadcast snapshot request messages with its id_colour to other nodes. When a white node receives a coloured snapshot request, changes its local colour to the request message colour, and forwards the message to its neighbour nodes. When a non-white node receives a different colour request messages to it local message, it knows there are more than one snapshot initiator on the network; the different initiator can be inferred from the colour of the request message. The coloured

node adds the id_colour of the other initiator to a list called border_list. After a node has received all requests from its incident edges, the colouring phase is finished. The node n_i sends its border_list to the node n_j that sent the first snapshot request message to n_i. When a node receives a border_list from its neighbours, it union the border_list with its own border_list and updates its own border_list with the resultant list. Finally, the initiator node that caused this request message gets this list. Figure 5-2 shows the phase when colouring has completed.

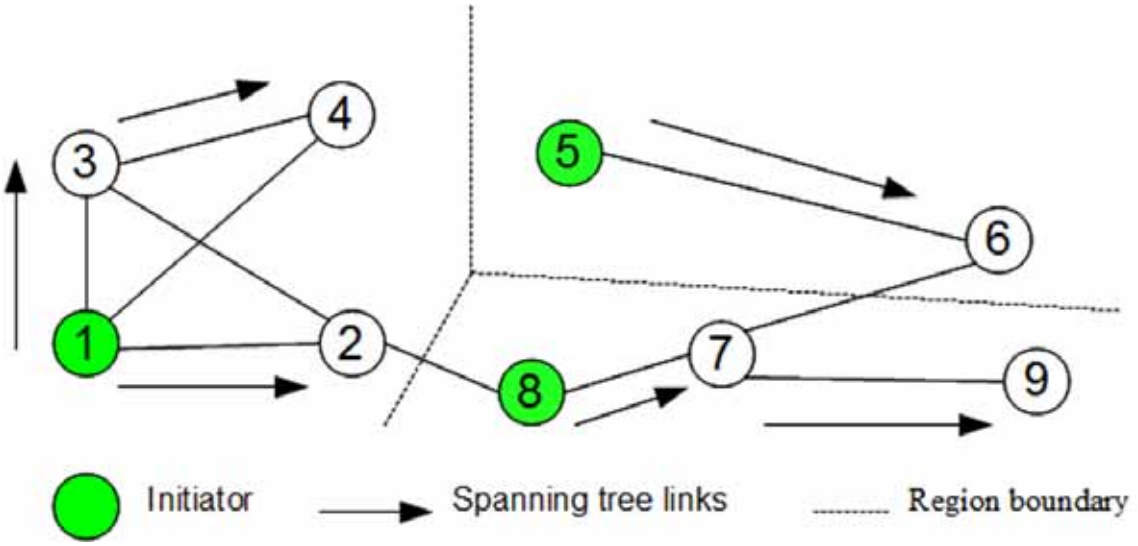


Figure 5-2 Colouring completed

In the colouring phase, the system is divided into different regions by different id_colours of initiators; the initiator knows its neighbouring initiators by border_list. The initiators exchange the partial snapshots of the nodes in their region. The exchanges occur in rounds, in each round of exchange, an initiator sends to its bordering initiator any new state information that it obtained during the previous round of message exchange. After the initiator state information exchange phase, each initiator has the consistent snapshot.

Figure 5-3 gives the example of the Spezialetti and Kearns' snapshot algorithm. There are three nodes on the network, process 1(P1), process 2(P2), and process 3(P3) are running on corresponding nodes. Node P2 initiates a snapshot collection by taking its local state, sends green mark to other nodes. Concurrently node P3 initiate a snapshot by taking its local state with red marker, send the marker to other nodes; the cut is the global snapshot. In Spezialetti and Kearns' snapshot algorithm P3 takes its snapshot in response to P2; P1 ignores P2's request. The global snapshot thus collected is shown by the cut.

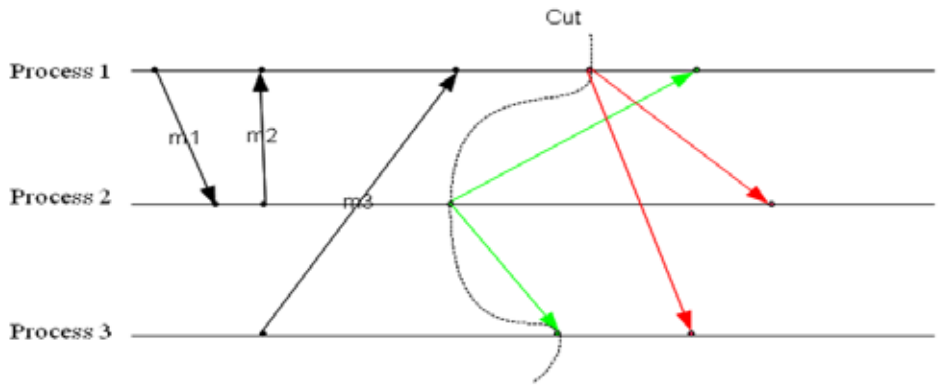


Figure 5-3 Spezialetti and Kearns' snapshot algorithm

All algorithms above are only suitable for the FIFO system, when the messages after the marker are sure to be sent. But for non-FIFO systems, the messages after marker cannot be sure to be sent. For example the CAN network is a priority based network communication channel, the message only can be sent by the node that has the highest priority; so it won't guarantee the message sends after the marker.

For the non-FIFO system different algorithms have been developed. Lai-Yang algorithm is the first non-FIFO snapshot system. It is based on the Chandy-

Lamport algorithm, and does not need markers to control other process to take snapshots.

5.3 Snapshot algorithm for non-FIFO

5.3.1 Lai–Yang algorithm

The Lai-Yang snapshot algorithm (T.H.Lai and T.H.Yang 1987) for non-FIFO does not send markers to request the other processes to record their local state; instead it colours messages red and white. If the sender has not recorded its state the sent message colour is white, if the sender has recorded its state the sent message colour is red. The messages are appended by the red or white colour. Processes are allowed to record their local state by itself (Sukumar Ghosh 2007). The algorithm can be stated as followed:

1. Every process is initially white and if it takes a snapshot, it turns to red.

2. Every message sent by a white (red) process is coloured white(red); so the white (red) message is sent before (after) the sender process record its local state.

3. Each process can record its local state at any time, but before possibly receives a red message.

   To make it possible, if the destination process (white) receives a red message, the destination process needs to record its local state before processing the message. By doing this, can make sure that there is no message sent after recording its local state, and also the marker is not required.

4. Each white process records a history of white messages that it sent or received by the channel.

5. If a process turns to red, then it sends all these white message histories and its local state to the initiator process.

6. The initiator process computes the channel state by evaluating **transit** (LS_i, LS_j), LS_i is local state of process *i*, LS_j is local state of process *j*.

   SC_{ij} = white message sent by P_i on C_{ij}

   White message received by P_j on

   $C_{ij} = \{m_{ij} | send(m_{ij}) \in LS_i\} - \{m_{ij} | rec(m_{ij}) \in LS_j\}$

Although the algorithm needs no control message, it needs to compute the channel state by the differences of histories of message and thus needs large storage for each process to store the message histories. The control message is used for the sake of recording a consistent snapshot, but the Lai-Yang algorithm does not use such control message, so the snapshot won't be consistent, unless the complete snapshot is taken. However the problem is if a process terminates, it will record its own state following its last action, but it won't tell the other processes, so the other processes won't record their state and the snapshot won't be consistent.

5.3.2 Mattern's algorithm

Mattern's algorithm (Friedemann Mattern 1993) works for both system FIFO and non-FIFO. The basic idea of Mattern's algorithm uses two colours to indicate whether a process has recorded its local state and whether a message is sent before or after the local state is recorded in a process. The algorithm is based on the vector clock that was discussed in section 4.4. The dimension of vector time equals the number of process, each dimension records the corresponding process local time. But the vector clock that is applied for Mattern's algorithm, is different

from previous vector clock; it is used to make sure all messages that are sent before the snapshot reach their destination before taking the snapshot. By doing this the vector time works as a counter, vector counter system allows to use a negative integer to count the time of message.

In the vector counter method any process p_i has a counting system for white messages (the message before the snapshot taking place). If p_i sent message to p_j ($i \neq j$) on the j-th component of a local vector V_i of length of number of processes, then $V_i[j] = V_i[j]+1$. If p_i receives a message, then $V_i[i] = V_i[i]-1$; so if all messages arrive at the destination process, the all components of vector clock should be zero.

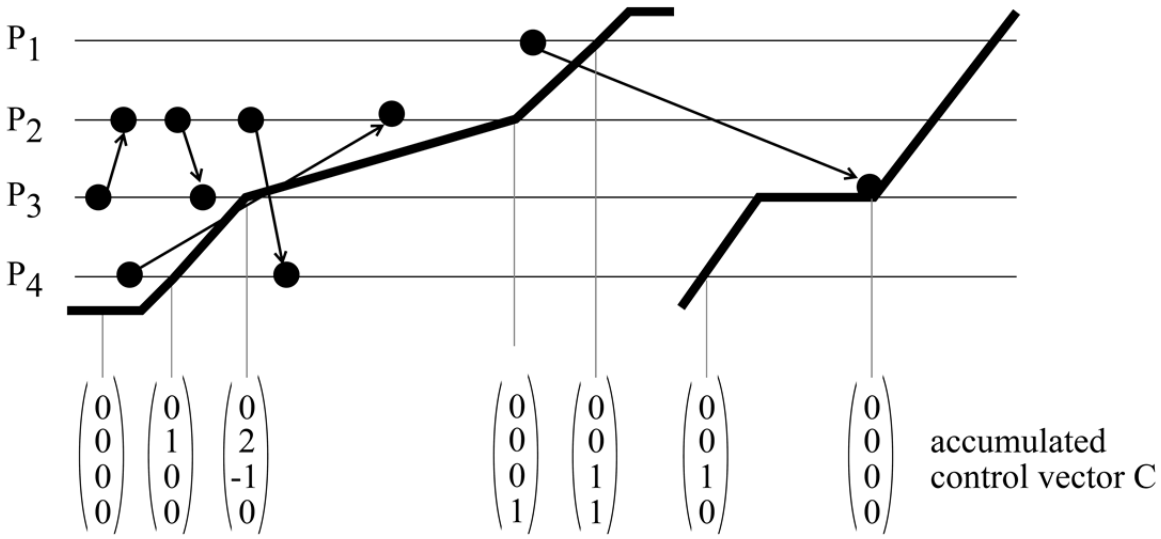


Figure 5-4 the vector counter method (Friedemann Mattern 1993)

Mattern's algorithm can be implemented with the following rules

1. At the beginning, every node colour is white. If a node sends (or receives) a white message, then it implements vector counting algorithm; message header

includes colour and timestamp, the timestamp is equal to or greater than the local time of sending node. If a red node sends a message, the red message timestamp updates to the minimum of this red message timestamp and the local time; the red message sent or received does not implement vector counter rules.

2. If an initiator node takes a snapshot, then the initiator colours itself as red, sends a marker to next node, the marker delivery is implemented as a token ring network; a marker contains minimum of local clocks, minimum timestamp, and local vector counter called count.

3. If a node n_i receives a marker, it takes a local snapshot, colours itself as red, the n_i waits $V[i]+count[i]\leq 0$ (where V is the local vector counter); by doing this can prove there is no white message that was sent to current node on the network, if the value smaller than zero the message can be ignored, otherwise it is not consistent; repeat rule 2.

4. If an initiator node n_{init} receives a marker, n_{init} wait until $V[init]+count[init]\leq 0$, if the count=0; then a value called Global Virtual Time (GVT) Approximation (GVT_approx) is generated. If the first round finishes with count $\neq$ 0, then the second round will start. But after second round marker pass the count is guaranteed to be zero vector and the GVT approximation is found. The main idea of GVT approximation is to use two cuts and to make sure that no messages cross both cuts. Hence, the minimum of the timestamps of all messages which cross the second cut can easily be determined by considering all messages which are sent between the two cuts. The snapshot

process is finished (Friedemann Mattern 1993). The message across the second cut is the state of the channel.

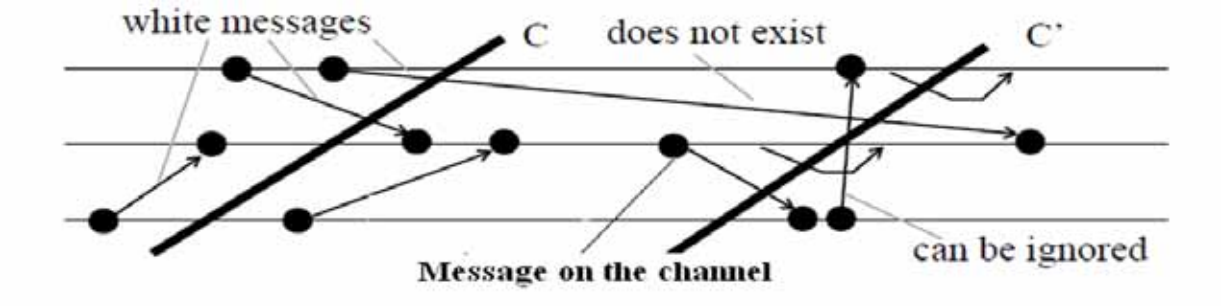


Figure 5-5 Example Mattern's algorithm

Figure 5-5 is an example of Mattern's algorithm; the last cut C' is the final snapshot, there are no message to cross the two cuts, the message received before C' but sent after C' is ignored, otherwise it is not consistent. The left part of C' is the global state of the system.

The Mattern's algorithm does not record the whole history of the channel state, but it needs to wait for the white message to finish delivery, so it delays the termination of the snapshot. It also uses a marker to check if any node terminated by accumulating the vector counter, a message counter per channel.

5.4 Comparison of snapshot algorithms

Algorithm	Feature
Chandy–Lamport(K.M.Chandy & L.Lamport 1985)	Original algorithm, only apply for FIFO channel. Using n×(n-1) markers generated on the network.
Spezialetti–Kearns(Madalene Spezialetti & Phil Kearns 1986)	Improvements over (K.M.Chandy & L.Lamport 1985): supports concurrent initiators, efficient assembly and distribution of a snapshot. Assumes bidirectional channels. O(e) messages to record, $O(rn^2)$ messages to assemble and distribute snapshot.
Lai–Yang(T.H.Lai & T.H.Yang)	Works for non-FIFO channels. Markers piggybacked on computation messages. Message history required to compute channel states.
Mattern(Friedemann Mattern 1993)	Works for both type of channels. Similar to (T.H.Lai & T.H.Yang) No message history required. Termination detection (e.g., a message counter per channel) required to compute channel states.

Table 5-1 snap shot algorithm comparison
n = # processes, u = # edges on which messages were sent after previous snapshot, e = # channels, d = diameter of the network, r = # concurrent initiators.

Chandy–Lamport algorithm is the first snapshot algorithm, it only suitable for the FIFO system, it consumes large channel resources to broadcast the marker as Table 5-1 shows, it does not include the algorithm to assemble the global state. Spezialetti–Kearns algorithm optimizes the Chandy–Lamport algorithm, it is not only sending the marker to record the state, it also uses messages to assemble the snapshots. The Chandy–Lamport algorithm only works for the FIFO channel, Lai–Yang is developed for non-FIFO channel and is easy to implement. It does not need send any mark on the channel, but each node has to store whole history of messages. The Mattern algorithm works for both types of channel, it does not

need to store the history of message, but for the termination detection it needs to compute the channel state.

5.5 Conclusions

This chapter introduced some snapshot algorithms that records the global state of the distributed system; these algorithms are very useful for the system analysis, testing or verifying properties associated with distributed executions.

In the automotive software testing, the global snapshot can be very useful to analyse the particular state that the system could be in the snapshot gives a global view of entire system state that includes states of every electronic control unit (ECU) and channel, so we can know what happened at the point of the snapshot by checking the value of each ECU variable and the messages on the network. However the snapshot algorithms only compute the instantaneous system global state, not a history of states. Some way of computing the whole history of system global states would be more useful for validating system behaviour over a period of time.

References

Friedemann Mattern. Efficient Algorithms For Distributed Snapshots And Global Virtual Time Approximation. 18, 423-434. 1993. *Journal Of Parallel And Distributed Computing*.

K.M.Chandy & L.Lamport. Distributed Snapshots: Determining Global: States Of Distributed Systems. 3. 1985. ACM Transactions On Computer Systems.

Madalene Spezialetti & Phil Kearns. Efficient Distributed Snapshots. 382-388. 1986. *Proceedings Of The 6th International Conference On Distributed Computing Systems*.

Rahul Garg, Vijay K.Garg, & Yogish Sabharwal. Efficient Algorithms For Global Snapshots In Large Distributed Systems. 1994. *IEEE Transactions On Software Engineering*.

Sukumar Ghosh. Distributed Systems An Algorithmic Approach. 2007. Taylor & Francis Group, LLC Chapman & Hall/CRC.

T.H.Lai & T.H.Yang. On Distributed Snapshots. **25**, 153-158. 1987. *Information Processing*
Letters.

Chapter 6 Global State Evaluation

6.1 Introduction:

Global Predicate Detection (GPD) is used to map all or part of the states that system goes through. A predicate is a requirement or standard, e.g. a comfortable room temperature is 20 Celsius to 29 Celsius, using this standard to check if the room temperature is comfortable. The predicate can be predefined by specification language (interval approach), or defined after correct executions (state approach). To define a predicate on the system state is very important to specify, observe, and detect the behaviour of a system. predicate specification and detection are very useful for distributed system analysis and testing.

In the automotive industry, networked ECUs are distributed systems and ECU integration testing is a hard and complex process during the automotive software development. GPD may help the ECU integration. As an example in the automotive industry, some cars have the auto-window wiper, when it is raining (the water toggles the rain sensor), the window wiper should work. As another example, an application might be interested in detecting the predicate engine temperature under 120 degrees Celsius and fuel use under 7 L/100 km.

Analysis of GPD is different from the global snapshot; the global snapshot is only one possible state that can be gone through during the whole execution of the system. GPD is the collection of whole execution path that has been executed; a snapshot is a one predicate value of a GDP. If the GDP is a map data structure, the snapshot is an element of the map; the map can be ordered by the time of the snapshot that has been taken, the logical time can be used as such time to order the snapshots in the GDP.

6.2 Stable and unstable Predicates

6.2.1Stable predicate

Predicates can be either stable or unstable, a stable predicate is a predicate that remains true once it becomes true (K.M.Chandy & L.Lamport 1985); there are two properties in the stable predicate, termination and deadlock.

6.2.1.1 Termination

The termination (Friedemann Mattern) predicate is stable (persistent), because if such state has been reached, it will never change. A process can be executed in two states, active and passive. An active state can be automatically changed to passive state (waiting), if there is no further work to do. A passive process can become active when it receives a message from another process. If a passive process receives a message it becomes active, it may send a message back to another process or processes; so the processes communicate by message passing. This is the basic communication style of the distributed system; all processes work together as one system, they interact with each other by message passing, an execution terminates if each process is passive, until it receives more messages. There are two conditions of the termination state, local and global:

- Local condition: Each process in a passive state
- Global condition: there is no message on the communication channel.

Assuming that the local condition can be characterized by a local state, the global condition can be characterized by a channel state; so it can be related with the system global snapshot, any channel state can be observed by the process local states that have been taken by two endpoints of the channel.

6.2.1.2 Deadlock

Deadlock (GARY S.HO and C.V.RAMAMOORTHY 1982) is another property of the stable predicate. Two or more processes are waiting for the resources of each other, but none of them can release the resource unless one of them release the resource to the other, such situation cause the deadlock; it just as the situation as “chicken or egg”.

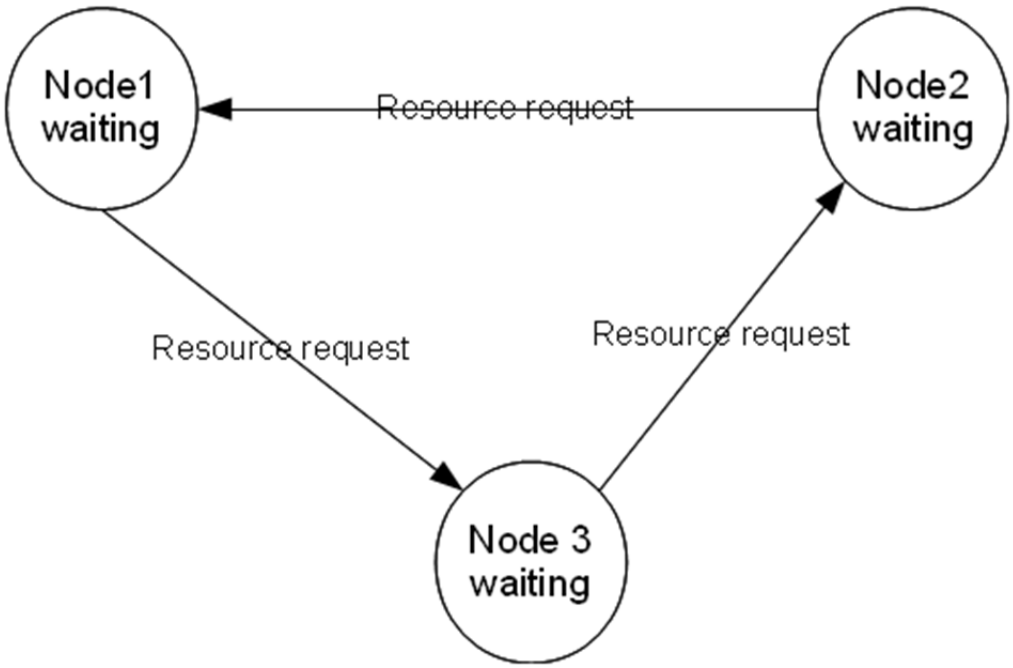


Figure 6-1 deadlock

Figure 6-1 shows a system that in the deadlock situation; node 1 waits on the resource from node 3, node 3 waits on the resource from node 2, node 2 waits on the resource from node 1. Each process in the node is blocked by waiting for the request from another node, also the deadlock process cannot receive a reply from some process(es); the deadlock just like a circular chain, if any link is broken, the deadlock is solved, or any node release the resources, the deadlock is also solved.

6.2.2Unstable predicate

The unstable predicate only holds the predicate instantaneously(Keith Marzullo and Gil Neiger 1991;Robert Cooper and Keith Marzullo 1991). There are some difficulties to detecting unstable predicates:

- The message transmitting times and scheduling of the various processes on the processors under various load conditions, they are all unpredictable; the execution is not deterministic, the distributed system may have gone through different global states; the predicate may be true in some executions, but fails in others.
- Instantaneous time in a distributed system is not available; if a predicate is true in a global state, it may not have held in the execution; if a predicate is true in the transmitting period, it may not be detected by a consistent snapshot. So the periodic monitoring of the execution is not enough.

There are two difficulties for the unstable predicate; one is the snapshot algorithm to be used to record the global state; another is the methodology to evaluate collected data.

To overcome these difficulties, two important observations can be made (Keith Marzullo & Gil Neiger 1991;Robert Cooper & Keith Marzullo 1991).

1. The entire execution monitoring is necessary, so all states that appeared in the execution can be examined.
2. The execution of a distributed system may go through different states every time it is executed; some predicate may be true in one execution but not in another; so it is very useful to define all the observations of the execution path not just one.

6.3 Possibly and definitely Predicates

Possibly (Φ): There exists a consistent observation of the execution such that predicate Φ holds in a global state of the observation.

Definitely (Φ): For every consistent observation of the execution, there exists a global state of it in which predicate Φ holds.

6.4 Relational Predicate

"A relational predicate is of the form $x_1+x_1+x_2+\ldots+x_n$ relop k, where each x_i is an integer variable on process p_i and relop $\in\{=, <, >, \leq, \geq\}$" (Neeraj Mittal and Vijay K.Garg 2001). The relational predicates are useful for detecting potential problems in a distributed system. For example, two processes: p_i and p_j each have a variable x and y, and communicate by passing messages, the predicate (x+y<9) may indicate a potential error (Alexander I.Tomlinson and Vijay K.Garg 1993).

The centralized relational GPD algorithms (Keith Marzullo & Gil Neiger 1991;Robert Cooper & Keith Marzullo 1991) for detecting possibly Φ and definitely Φ are based on the same data structure.

The data structure is built as lattices, the lattice is a possible execution path, ordered by the vector time and every subsequent point in the path is increased by one. The lattices are arranged by level that is the sum of components of the vector time, all states in the same level will not affect each other, because they are the potential states that are reached from the previous level, so they are concurrent.

Figure 6-2 is the example of the lattices of global predicate states, S is the global state the execution is going through, and the numbers on the right corner is the vector clock of that global state. This example only has two processes, so the vector time only has two components.

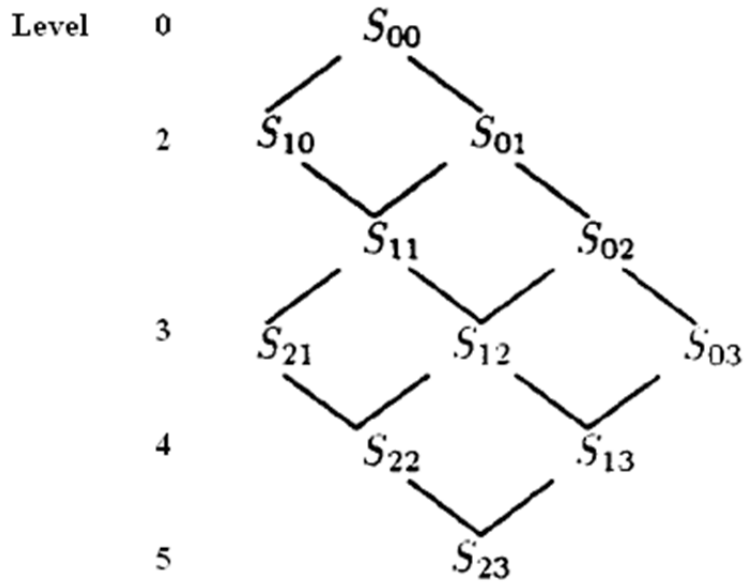


Figure 6-2 the lattices of global predicate state

The centralized algorithms are able to assemble the local state to the global state, order the global state into the lattices. There is a central process P_0 and each local process P_i sends its local state trace, each state with its vector time, to P_0; P_0 contains n queues, $Q_1 \ldots Q_n$. Q_i contains P_i's states trace; each local state of a process are stored in the queue. Figure 6-3 illustrates the queues where P_i stores each local state trace of each process.

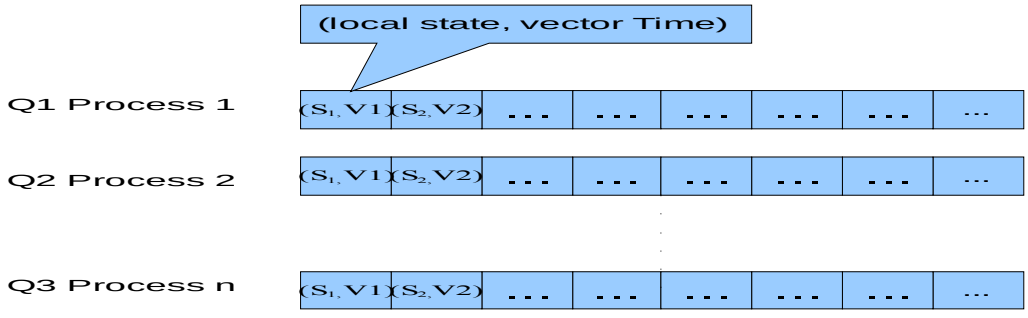


Figure 6-3 Local trace of states in the queues of central process

A global state is assembled from the local states; all these local states are selected from one element of each queue; but which local state should be selected from

each queue? The question can be answered by the following rules, based on the vector time. Each local state in the queue is labelled with its vector time $V(S_i)$, S_i is a local state of P_i. A valid global state has to satisfy (Robert Cooper & Keith Marzullo 1991, p170)

$$\forall\ i,j : 1 \leq i,j \leq n : V(S_i)[i] \geq V(S_j)[i]$$

This condition states that a message cannot be received before it is sent. The newest update of the vector time is only known by the corresponding process itself, other processes only hold the earlier time than or the same time as the process.

For each local state S_i of a process P_i there is exists a minimum global state $S_{min}(S_i)$ that contains S_i and a maximum global state $S_{max}(S_i)$ that contains S_i. The global states are (Robert Cooper & Keith Marzullo 1991, p170):

$S_{min}(S_i)=(S_1, S_2, \dots, Sn) : V(S_j)[j]=V(S_i)[j]$ and

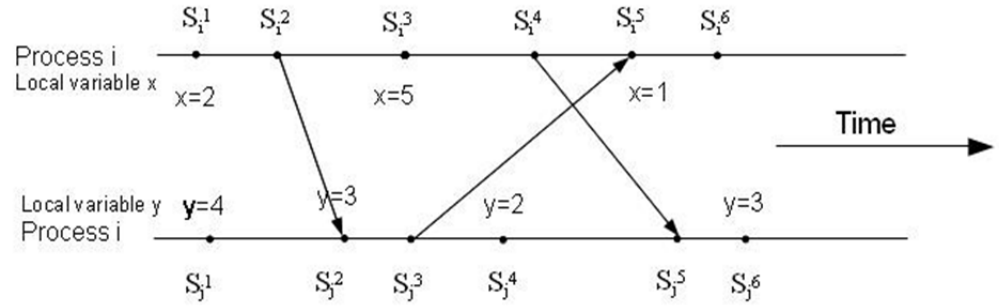
$S_{max}(S_i)= (S_1, S_2, \dots, Sn) : V(S_j)[i] \leq V(S_i)[i] \wedge \forall\ S'_j : S_j \rightarrow S'_j\ V(S'_j)[i] > V(S_i)[i]$

These two rules constrain the selection of the levels that S_i occurs; the minimum level containing S_i is very easy to compute, it is the sum of components of the vector timestamp $V(S_i)$; for each sequence Q_i , P_0 can construct the set of states at each level; the sum of the components of timestamp of the last element of Q_i is the last level. For any level that is greater than the last level ($S_{max}(S_i)$), P_0 removes S_i from Q_i.

Given the states of level lvl, the set of states at level lvl+1 can be constructed as follows; for each global state $GS(S_1, S_2, \dots, S_n)$, construct the n global state $(S_{1+1}, S_2, \dots, S_n)$ $(S_1, S_{2+1}, \dots, S_n)\dots(S_1, S_2, \dots, S_{n+1})$.

Figure 6-4 gives the example how to construct the lattices from the corresponding execution.

(a)



(b)

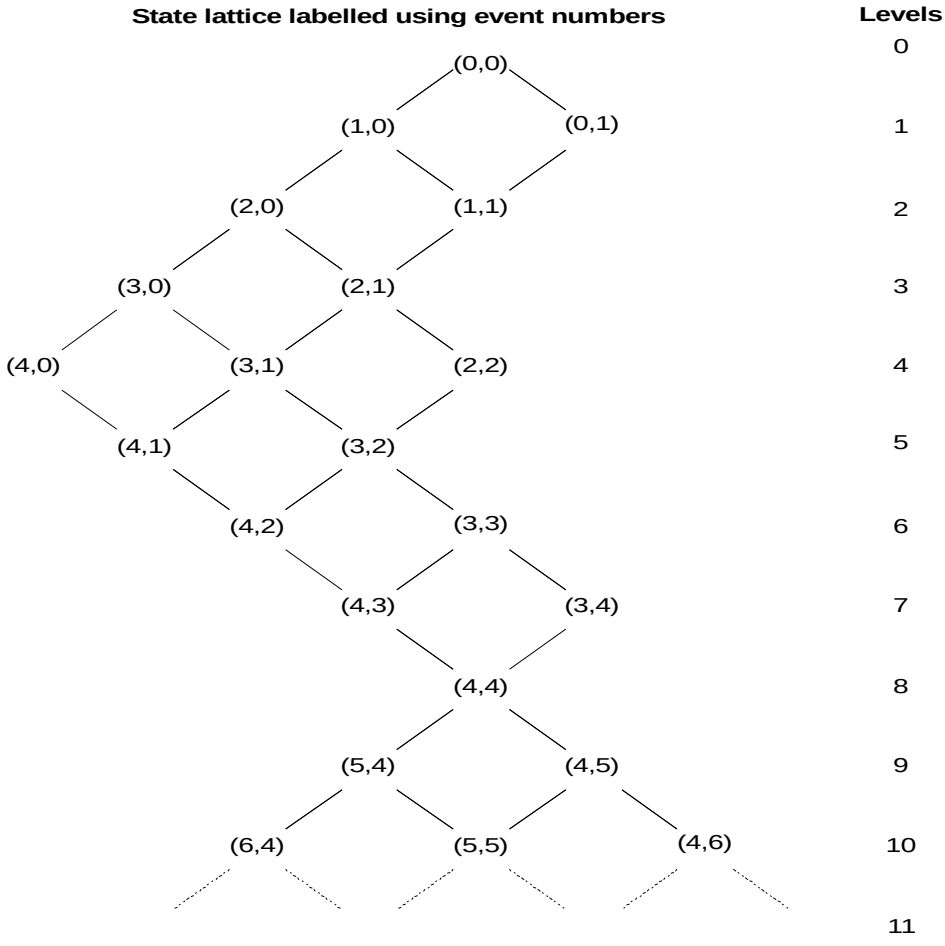


Figure 6-4 Example to show the states build into the lattices, the level to the corresponding lattices. (a) Corresponding state lattice of the execution of figure. (b) the state lattice for the execution.

6.4.1 Algorithms to detect possibly predicate and definitely predicate

After the state lattice is built, detecting predicate can proceed. The algorithm for detecting a relational predicate by examining the state lattice can be describe as following:

Variables:

Set of global states current $_{\Phi}$, next $_{\Phi}$ ← GS(0,0,...,0)

int level ← 0

Possible(Φ):

While (no state in current $_{\Phi}$ satisfies Φ) **do**

If (current $_{\Phi}$={final state}) **then return** false;

lvl ← lvl+1;

current $_{\Phi}$ ← {states at level lvl};

return true.

Definitely(Φ):

remove from current $_{\Phi}$ those states that satisfy Φ

lvl ← lvl+1;

while (current $_{\Phi}$ not empty) **do**

next $_{\Phi}$ ← {states of level lvl reachable from a state in current $_{\Phi}$ };

remove from next $_{\Phi}$ all the states satisfying Φ;

if next $_{\Phi}$ = {final state} **then return** false;

lvl ← lvl+1;

current $_{\Phi}$ ← next $_{\Phi}$;

return true.

possibly(Φ): To detect possibly predicate, an exhaustive search of the state lattice is performed; any one state that satisfies Φ the search can be terminated; the algorithm examines the lattice level by level. It starts from level 0, ends at the final state; each level is examined to find a state that satisfiesΦ; if such state exists the algorithm terminates.

Definitely(Φ): For definitely to be true, there should exist a set of states that satisfy Φ, every path of execution going through one of these states. It is not necessary that all these states are at the same lattice level. Figure 6-5 is the example of the set of states that are not at the same level, but all the states in the set satisfy the predicate and every execution path goes through one of these states(4,6), (5,5), (6,5), (7,4).

Because definitely (Φ) may be true but the sets of states may not at the same level, the Possibly (Φ) algorithm approach cannot be applied to the Definitely (Φ) predicate; definitely approach detects the states that not satisfy predicate Φ (satisfy $\neg\Phi$); rather than track the state in which Φ is true, it tracks the states in which Φ is not true, the set of states tracked at different level should be reachable from the previous level; variable next $_{\Phi}$ is used to check any states in the next level that do not satisfy predicate; the states that do not satisfy predicate are stored into the next $_{\Phi}$, next $_{\Phi}$ go through all the level from initial to final, if the next $_{\Phi}$ is empty the algorithm terminated successfully, otherwise it terminates unsuccessfully.

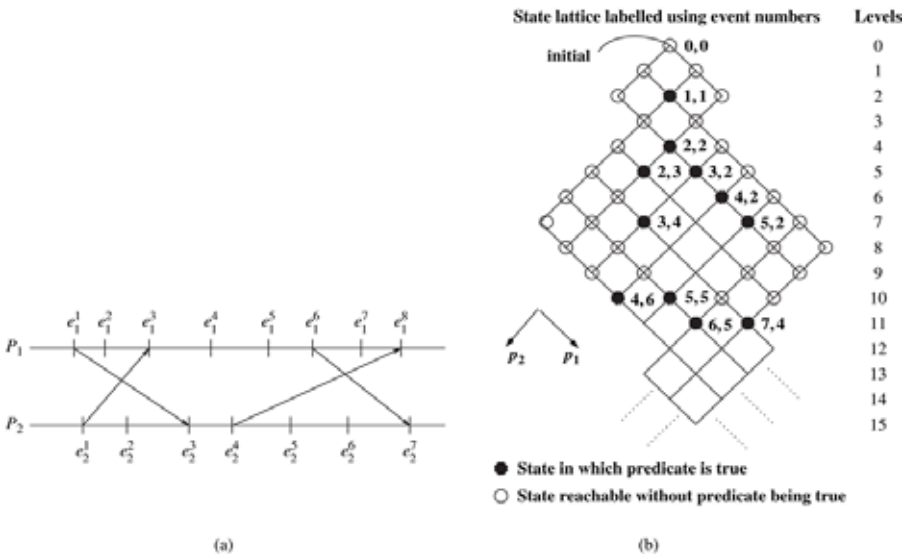


Figure 6-5 Example to show that states in which Definitely Φ is satisfied need not be at the same level in the state lattice. (a) Execution. (b) Corresponding state lattice.

6.5 Conjunctive Predicate

Up to now, the predicates introduced are relational; the predicate can be specified in the system, after all execution paths are constructed. Another predicate is called a conjunctive predicate, where a predicate Φ will be given first, and the states that satisfy Φ are recorded. A predicate Φ is a conjunctive predicate if and only if Φ is the logical "AND" of local predicates. It can expressed as the conjunction $\wedge_{i,j \in n} \Phi_i$, where Φ_i is a predicate local to process i, n is the total number of process. The predicate of interest can be modelled as a conjunctive predicate; there are two main algorithms for the conjunctive predicate: centralized algorithm (Figure 6-6) and distributed algorithm (Figure 6-7). The difference between centralized and distributed algorithm is that the centralized algorithm support offline GDP evaluation and the predicates can be specified after system execution; the distributed algorithm evaluates the predicate in the real-time and the predicate has to be specified ahead. This section will describe the algorithms for conjunctive predicate.

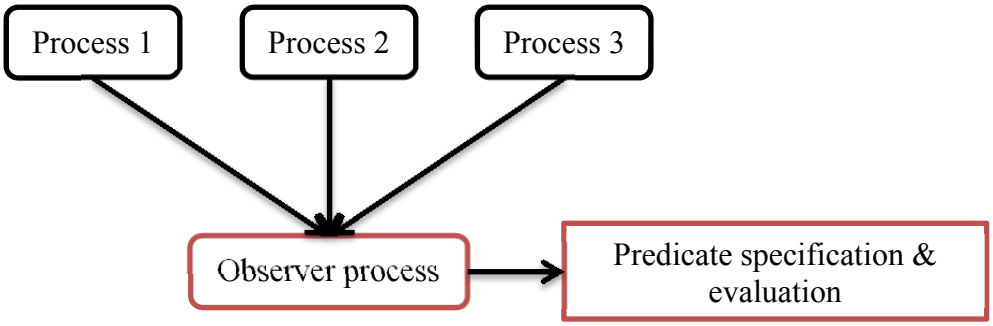


Figure 6-6 centralized algorithm

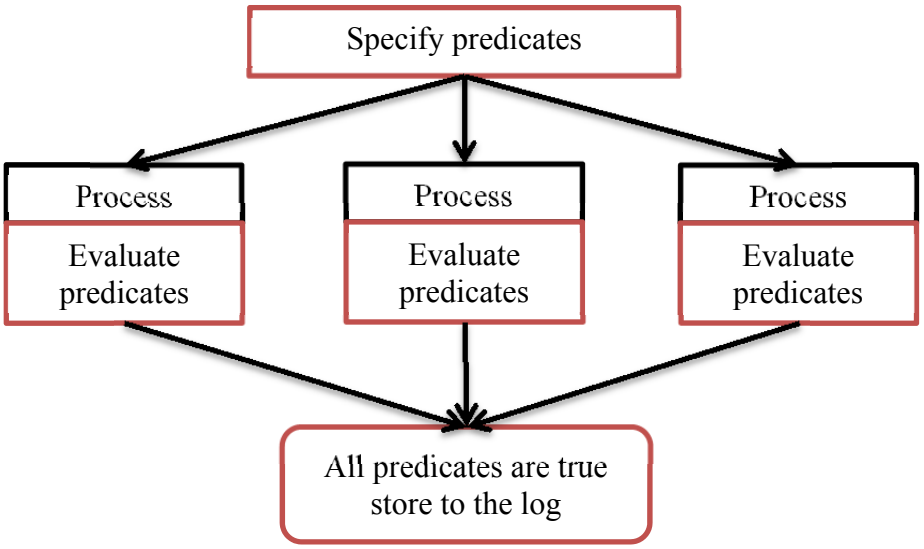


Figure 6-7 distributed algorithm

6.5.1 Interval-based centralized algorithm for conjunctive predicate

In the interval based approach, the local predicate changes between false and true; for some periods the local predicate is true in a process, some periods it is false; the local predicate shifts during the execution, and is illustrated in Figure 6-8.

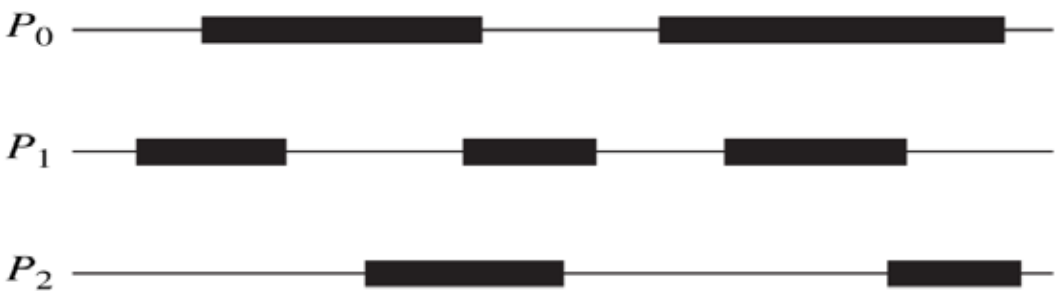


Figure 6-8 for a conjunctive Predicate the shaded durations indicate the periods when the local Predicates are true.

For two processes P_i and P_j, when the local predicate Φ_i and Φ_j are true, the intervals of these processes are denoted X_i and Y_j. Let the start and end of interval X be denoted as min(X) and max(X), for interval Y is the same. Assume the global state predicate is defined on these two processes, then Definitely (Φ) and possibility (Φ) can be defined as:

Equation 6-1 $\quad$ Definitely(Φ):$\min(X) \prec \max(Y) \wedge \min(Y) \prec \max(X)$;

Equation 6-2 $\quad$ $\neg$Possible(Φ) :$\max(X) \prec \min(Y) \vee \max(Y) \prec \min(X)$;

where $\prec$ means an irreflexive partial ordering representing the causality relation on the event set(Ajay D.Kshemkalyani 2003), for example if event $e \prec e' \Leftrightarrow T(e) < T(e')$, T is the timestamp. Figure 6-9 shows the condition for two processes.

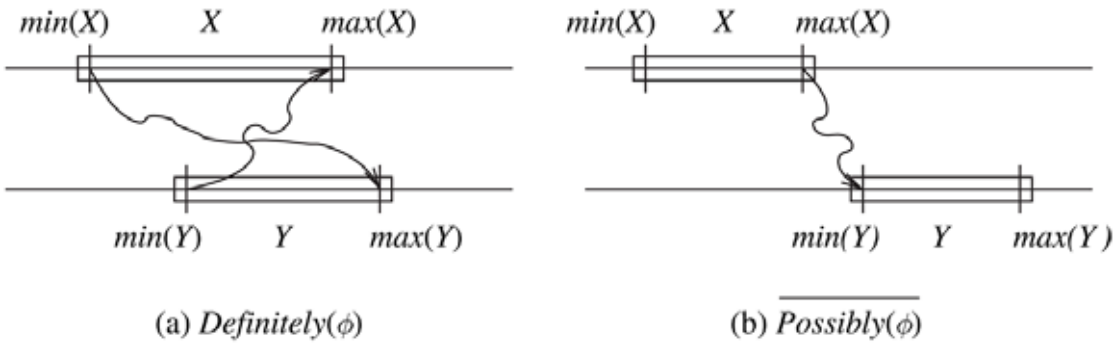


Figure 6-9 Illustrating conditions for Definitely(Φ) and $\neg$Possible(Φ), for two processes.

The conjunctive predicate for more than two processes (Ajay D.Kshemkalyani 2003) are defined as

Equation 6-3 Definitely(Φ) if and only if $\wedge_{i,j\in n}$ Definitely($\Phi_i \wedge \Phi_j$);

Equation 6-4 Possibly(Φ) if and only if $\wedge_{i,j\in n}$ Possibly($\Phi_i \wedge \Phi_j$);

Interval-based centralized algorithm runs the algorithm on a central server P_0 to monitor possibly or definitely conjunctive predicate Φ (Punit Chandra and Ajay D.Kshemkalyani 2005;Vijay K.Garg and Brian Waldecker 1994;Vijay K.Garg and Brian Waldecker 1996). When a local predicate is true, the process can send its vector timestamp of start and end events of an interval to P_0, the interval is part of the log. Another element of the log is a queue of events in the interval, called the process log; P_0 stores the log of a process to a queue; each queue contains a set of intervals of one process. Figure 6-10 is the data structure of the data structure for an interval queue of P_0.

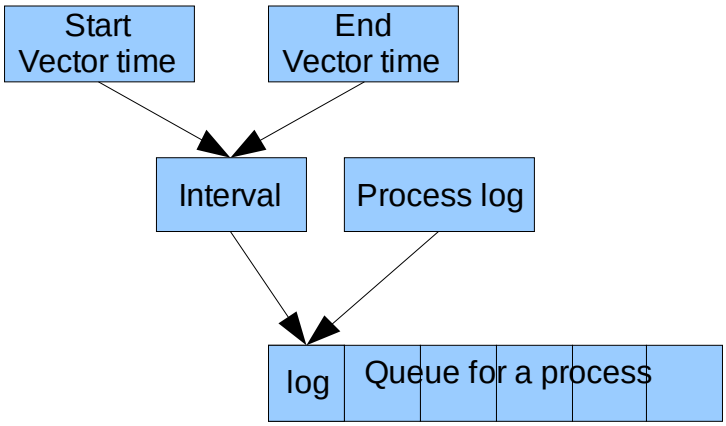


Figure 6-10 data structure for an interval queue of central process P_0

If any message send or receive event between start of previous interval and the end of later interval, then an interval needs to be sent to central process, each

execution need send most 4 messages, two from sender and another two from receiver.

There are two queues in the algorithm, updatedQueues and newUpdatedQueues. The updatedQueues stores the indices of the queues whose heads got updated; the newUpdatedQueues is a temporary variable to update updatedQueues. There are two situations to update the queue of a log of a process; when a new log is added to the head of the queue, or the head of the queue is deleted, as determined by Equation 6-1. After the queue is updated, the new head is the candidate log, the interval of the log is the candidate interval. Each new candidate interval is examined with the head of all other queues by Equation 6-1. In each comparison, if it does not satisfy the Equation 6-1, one of the two intervals examined is marked for deletion and the queue is updated by doing the deletion.

queue of Log: Q_i, Q_2, … $Q_n \leftarrow \perp$
set of int: updatedQueues, newUpdatedQueues $\leftarrow$ {}
On receiving interval from process P_z at P_0:
(1) Enqueue the interval onto queue Q_z
(2) if (number of intervals on Q_z is 1) then
(3) updatedQueues $\leftarrow$ {z}
(4) while (updatedQueues is not empty)
(5) newUpdatedQueues $\leftarrow$ {}
(6) for each i $\in$ updatedQueues do
(7) if (Q_i is non-empty) then
(8) X $\leftarrow$ head of Q_i
(9) for j = 1 to n do
(10) if (Q_j is non-empty) then
(11) Y $\leftarrow$ head of Q_j
(12) if ($\neg$(min(X) $\prec$ max(Y))) then // Definitely
(13) newUpdatedQueues $\leftarrow$ {j}$\cup$newUpdatedQueues
(14) if ($\neg$ (min(Y) $\prec$ max(X))) then // Definitely
(15) newUpdatedQueues $\leftarrow$ {i}$\cup$newUpdatedQueues
(12') if ($\neg$ (max(X) $\prec$ min(Y))) then // Possibly
(13') newUpdatedQueues $\leftarrow$ {i}$\cup$newUpdatedQueues
(14') if ($\neg$ (max(Y) $\prec$ min(X))) then // Possibly
(15') newUpdatedQueues $\leftarrow$ {j}$\cup$newUpdatedQueues
(16) Delete heads of all Q_k where k $\in$ newUpdatedQueues
(17) updatedQueues $\leftarrow$ newUpdatedQueues
(18) if (all queues are non-empty) then
(19) solution found. Heads of queues identify intervals solution.

$\perp$ means empty

The algorithm above is a centralized algorithm, it detects a conjunctive for possibly or definitely predicate; lines 12-15 are for definitely predicate, lines 12'-15' is for possibly.

The set updatedQueues stores the indices of all the queues whose heads get updated. In each iteration of the while loop, the index of each queue whose head is updated is stored in set newUpdatedQueues (lines 12–15 or 12'–15'); In lines 16 and 17, the heads of all these queues are deleted and indices of the updated queues are stored in the set updatedQueues. Thus, an interval gets deleted only if it cannot be part of the solution. Now observe that each interval gets processed unless a solution is found using an interval from each process. According to Def5.5.3 and Def5.5.4, if each queue is not empty and their head cannot be deleted, then the set of logs at the head of each queue forms the global state that satisfy the conjunctive predicate.

6.5.2 Distributed algorithms for conjunctive predicate

6.5.2.1 Distributed state based token algorithm for possibly conjunctive predicate

In the distributed state based token algorithm, the queue Q_i stores the local vector times of process P_i. Q_i is maintained locally at P_i; there is a token passing through whole network. The token contains a vector time (Vtime) which is the newest update of each process and a set of Boolean values (Valid) that are flag for the validation of local state of a process. If the local state is validated (predicate is true) the value becomes to true. The data structure for a token can be expressed as:

```
struct token{
interger: Vtime[1…n];
boolean: Valid[1…n];
                    }
```

where n is the total number of process.

When algorithm starts, program initialized as:

queue of array of integer: $Q_i \leftarrow \perp$;

Token can be randomly initialized by a process;

when a process P_i receives a token it does following:

1.	while(*token.Valid[i]*)=0 do	Check if the current local state of process P_i is validated.
2.	await (Q_i to be nonempty)	Waiting till at least one vector timestamp is stored into the P_i
3.	if((head(Q_i))[i]>*token.Vtime[i]*) then	Earliest timestamp of P_i may be part of vector timestamp of the token, it depends on if the earliest timestamp is greater than the timestamp of token.
4.	*token.Vtime[i]* ← (head(Q_i))[i]	
5.	*token.Valid[i]* ← 1;	
6.	else dequeue head(Q_i);	Delete the inconsistent vector timestamp
7.	for j=1 to n (i≠j) do	Checking if the timestamps of other process of the local vector time consist with the vector timestamps of token.
8.	if i≠j and (head(Q_i))[j] *token.Vtime*[j] then	
9.	*token.Vtime*[j] ← (head(Q_i))[j];	
10.	*token.Valid[j]* ← 0;	
11.	dequeue (head (Q_i));	
12.	if for some k, token.valid[k]=0 then	
13.	send token to p_k;	
14.	else return (1);	

When a process P_i can receive a token, only *token.Valid[i]=0*. line 3 to 6 compares the ith vector time of earliest timestamp of Q_i with the *token.Vtime[i]* (the earliest timestamp in Q_i is head(Q_i)); if head(Q_i) greater than *token.Vtime[i]*, then the *token.Vtime[i]* updated to *head(Q_i)[i]*, the *token.Valid[i]* set to true, the next step need to check if other vector time components of *head(Q_i)* consistent with corresponding components of vector time of token, line 7 starts such a checking loop; if line 8 is true the state of P_j is not consistent (Figure 6-11(a)), *token.Valid[j]* is reset, the token is sent to P_j before termination of the algorithm

and P_j needs to check its Q_j that is consistent with all the other states in *token.Vtime*; otherwise the state of P_j is consistent (Figure 6-11(b)).

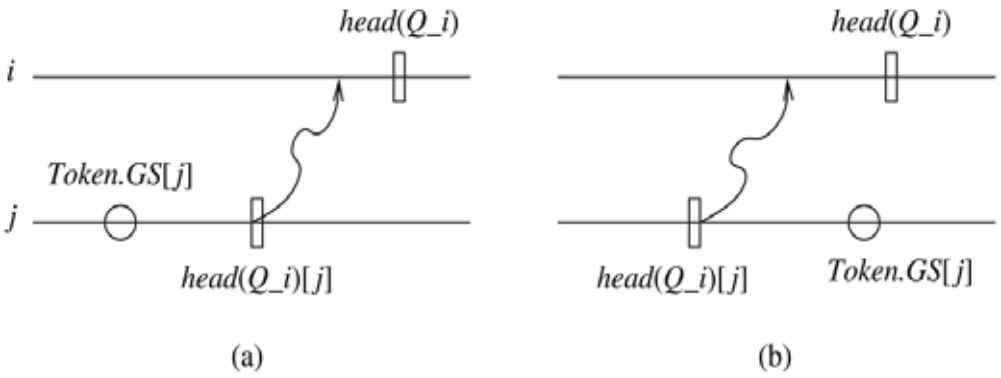


Figure 6-11 two possibilities assigns *head(Q_i)[i]* to a *token*

If all values of Valid of the token, then a solution is found; otherwise the algorithm is going to repeat, the code goes back to line 1 as line 14 stated.

6.5.2.2 Distributed interval-based token algorithm for Definitely conjunctive predicate

The interval-based token algorithm for definitely conjunctive predicate is based on the "Distributed algorithm to detect strong conjunctive Predicates" (Punit Chandra and Ajay D.Kshemkalyani 2003) . Define $I_i \downarrow I_j$ as $\min(I_i) \prec \max(I_j)$

Problem statement. In a distributed execution, identify a set of intervals I containing one interval from each process, such that

(i) the local predicate Φ_i is true in $I_i \in I$, and

(ii) (ii) for each pair of processes P_i and P_j , Definitely(Φi,j) holds, i.e., $I_i \downarrow I_j$ and $I_j \downarrow I_i$.

Before explaining the algorithm, the data types that are used in the algorithm should be stated. The type of Log contains start (V_i^-) and end (V_i^+) vector

times of the interval. Each process has a queue to store the logs. An interval Y at P_j is deleted if on comparison with some interval X on P_i, $\neg X \mathrel{|}\!\rightarrow Y$, i.e., $V_i^-(X) > V_j^+(Y)[i]$. Thus the interval(Y) being deleted or retained depends on its value of $V_j^+(Y)[i]$. The value $V_j^+(Y)[i]$ changes only when a message is received. Hence an interval needs to be stored only if a receive has occurred since the last time a Log of a local interval was queued. The Table 6-1 shows the local process data type used in the algorithm.

```
type Log
        start: array[1 . . .n] of integer;
        end: array[1 . . .n] of integer;
type Q: queue of Log;
When an interval begins:
Log_i.start ← V_i^-
When an interval ends:
Log_i.end ← V_i^+
if (a receive event has occurred since the last time
      a Log was queued on Q_i ) then Enqueue Log_i on to the local queue Q_i .
```

Table 6-1 Tracking intervals locally at process P_i.

There are three types of message in the algorithm; request message of type REQUEST, reply message of type REPLY, token message of type TOKEN, they are denoted as REQ, REP, and T, respectively Table 6-1 shows the message type for the algorithm.

```
type REQUEST          //used by P_i to send a request to each P_j
start: integer;       //contains Logi.start[i] for the interval at the queue head of P_i
end: integer;         //contains Logi.end[j] for the interval at the queue head of P_i , when sending to P_j
type REPLY            //used to send a response to a received request
updated: set of integer;      //contains the indices of the updated queues

type TOKEN            //used to transfer control between two processes
updatedQueues: set of integer; //contains the index of all the updated queues
```

Table 6-2 Message Type

```
1   Process Pi initializes local state
      Qi is empty.
2   Token initialization
      A randomly elected process Pi holds the token T.
      T.updatedQueues ← {1,2,…,n}.
3   RcvToken: When Pi receives a token T:
      Remove index i from T.updatedQueues
      wait until (Qi is nonempty)
      REQ_start ← Logi.start[i], where Logi is the log at head of Qi
      for j = 1 to n do
        REQ_end ← Logi.end[j]
        Send the request REQ to process Pj
      wait until (REPj is received from each process Pj)
      for j = 1 to n do
        T.updatedQueues←T.updatedQueues ∪REPj.updated
      if (T.updatedQueues is empty) then
        Solution detected. Heads of the queues
       identify intervals that form the solution.
      else
        if (i ∈ T.updatedQueues) then
          dequeue the head from Qi
        Send token to Pk where k is randomly
          selected from the set T.updatedQueues.
4   RcvReq: When a REQ from Pi is received by Pj :
      wait until (Qj is non-empty)
      REP.updated ← Ø
      Y ← head of local queue Qj
      V_i^-(X)[i] ← REQ.start and V_i^+(X)[j] ← REQ.end
      Determine X ↳ Y and Y ↳ X
      if(¬(Y ↳ X)) then REP.updated← REP.updated∪ {i}
      if(¬(X ↳ Y)) then
        REP.updated ← REP_updated ∪ {j}
        Dequeue Y from local queue Qj
      Send reply REP to Pi.
```

Table 6-3 Distributed algorithm to detect Definitely(Φ).

In the algorithm only the token-holder can send REQs and receive REPs to all other processes (line 3f), *Logi.start[i]* and *Logi.end[j]* for the interval at the head of the queue Q_i are piggybacked on the request REQ sent to process P_j lines (3c–3e). On receiving a REQ from Pi , process Pj compares the piggybacked interval X with the interval Y at the head of its queue Qj (line 4e). The comparisons between intervals on process Pi and Pj can result in these outcomes. (1) Definitely(Φi,j) is satisfied. (2) Definitely(Φi,j) is not satisfied and interval X can be removed from the queueQ_i . The process index i is stored in REP.updated (line 4f). (3) Definitely(Φi,j) is not satisfied and interval Y can be removed from

the queue Q_j . The interval at the head of Q_j is dequeued and process index j is stored in REP.updated (lines 4g, 4h). Note that outcomes (2) and (3) may occur together. After the comparisons, P_j sends REP to P_i. Once the token-holder process P_i receives a REP from all other processes, it stores the indices of all the updated queues in the set T.updatedQueues (lines 3h, 3i). A solution, identified by the set I formed by the interval Ik at the head of each queue Q_k , is detected if the set updatedQueues is empty. Otherwise, if index i is contained in T.updatedQueues, process P_i deletes the interval at the head of its queue Q_i (lines 3m, 3n). As the set T.updatedQueues is non-empty, the token is sent to a process selected randomly from the set (line 3o). The correctness of the algorithm is based on Equation 6-1 and Equation 6-3. The following observations can be made:

- If Definitely(Φi,j) is not true for a pair of intervals X_i and Y_j , then either i or j is inserted into T.updatedQueues.
- An interval is deleted from queue Q_i at process P_i if and only if the index i is inserted into T.updatedQueues.
- When a solution I is detected by the algorithm, the solution is correct, i.e., for each pair $P_i,P_j \in N$, the intervals I_i = head(Q_i) and Ij = head(Q_j) are such that $I_i \not\hookrightarrow I_j$ and $I_j \not\hookrightarrow I_i$ (and hence by Equation 6-1 and Equation 6-2, Definitely(Φ) must be true).
- If a solution I exists, i.e., for each pair $P_i,P_j \in N$, the intervals I_i, I_j belonging to I are such that $I_i \not\hookrightarrow I_j$ and $I_j \not\hookrightarrow I_i$ (and hence from Equation 6-1 and Equation 6-2, Definitely(Φ) must be true), then the solution is detected by the algorithm.

6.6 Predicate detection in automotive system

Comparing to the desk computer, the automotive distributed systems have less CPU power, less memory, and different network protocols. Therefore to apply GPD on the automotive system has to satisfy the following requirement:

- No interference with application software so as not to effect its real-time characteristics.
- The local state cannot be recorded into the local ECU memory.
- As less as possible to use the processor power to implement the GPD functions.
- CAN bus is non-FIFO and event triggered network (the message are transmitted in priority order.).

The above requirements for the automotive system are used to compare and contrast the alternative GPD options

6.6.1 Distributed algorithm vs. centralized algorithm

The distributed GPD algorithms need to specify the predicate ahead and each node only evaluates its own predicates (it only knows its local variable's value). Therefor the specified predicate cannot express the logical relationship between the variables that are in the different nodes, e.g. x in node1 and y in node2, for the distributed algorithm it is impossible to give the predicate like "x=y", because node1 does not know node2's variable. Each node has to evaluate the predicates locally (more process are needed).

The centralized GPD algorithms use extra process to observe each local process. The predicate can be specified after the system execution and the evaluation can

be done offline. It does not use much local processor power. Almost all heavy work can be done by the extra process.

6.6.2 Conjunctive vs. relational predicate

The conjunctive predicates need to specify the predicate ahead. Each node need to evaluate its own local predicates, the evaluation results are connected by logical "AND". If the conjunction is true, the local states construct the validated global state for the predicate. The global state is stored in a log. This way makes the storage is smaller, due to it only logs the validated predicate states. However it does not give the whole execution trace.

The relational predicate can be done by the offline. All global states are captured and the execution lattice is built up. The predicates can be specified after the execution. It gives a global over view of the execution; however comparing to the conjunctive predicate it consumes more memory.

6.6.3 GPD algorithm choice

Finally putting all together the requirements of the automotive systems, the comparison of distribute GPD algorithm and centralized GPD algorithm, and the comparison of conjunctive predicate and relational predicate, the centralized and relational algorithm is the better choice. They do not consume too much local process power, an extra process can be used to do the heavy work; this extra process may be done by a desktop computer which is much more powerful than an ECU. Also the global state can be recorded into the hard disk of the desktop computer; it won't consume any storage of the ECU. Because the GPD algorithms do not record the communication channel states, the network protocol is not big issue for it.

6.7 Conclusion

The global state capture is a fundamental problem in the asynchronous distributed system; this has been discussed in chapter 4. The extension of this problem is to observe global states that satisfy a given predicate of variables of the system. If the Predicates remain true once they become true, it is called a stable predicate; dead lock and termination detection are based on the settable predicate detection. Another predicate is unstable predicate which is very hard to detect, because the values of variables that make the predicate true can change and falsify the predicate.

The unstable predicate can be defined as possibly and definitely, Possibly means the execution may go through the global states that satisfy the predicate, Definitely means all executions must go through a global state that is in the set of global states that satisfy the predicate.

There are two ways to specify predicates, relational and conjunctive. Relational predicate detection collects entire execution states, and uses these global states to build execution lattices, the predicate can be specified after the lattices are built. From the execution lattices, the definitely and the probably predicate can be found. The conjunctive predicate detection needs to specify the Predicates ahead of execution; only the global states that satisfy this conjunctive predicate can be captured.

The global predicate detection could be applied to automotive distributed system testing, using the predicate detection to verify if the system is in the right states given by predicate. In order find the failure states of the system, the failure variable can be used as Predicates. For some important issues of a car such as

security, e.g. the brake or airbag. Predicate detection can be used to detect unsafe system states. Finally a centralized relational predicate algorithm is chosen to apply on the automotive distributed system testing.

References

Ajay D.Kshemkalyani. A Fine-Grained Modality Classification for Global Predicates. 14, 807-816. 2003. IEEE Transactions.

Alexander I.Tomlinson & Vijay K.Garg. Detecting relational global predicates in distributed system. 28[12]. 1993. ACM New York, NY, USA.

Friedemann Mattern. Algorithms for distributed termination detection. 2, 161-175. Distributed Computing.

GARY S.HO & C.V.RAMAMOORTHY. Protocols for Deadlock Detection in Distributed Database Systems. 8. 1982. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

K.M.Chandy & L.Lamport. Distributed Snapshots: Determining Global: States of Distributed Systems. 3. 1985. ACM Transactions on Computer Systems.

Keith Marzullo & Gil Neiger. Detection of Global State Predicates. [LNCS 579], 254-272. 1991. Proceedings of the 5th Workshop on Distributed Algorithms.

Neeraj Mittal & Vijay K.Garg. On Detecting Global Predicates in Distributed Computations. 3-10. 2001. International Conference on Distributed Computing Systems.

Punit Chandra & Ajay D.Kshemkalyani. Distributed algorithm to detect strong conjunctive predicates. 87, 243-249. 2003. *Information Processing Letters*.

Punit Chandra & Ajay D.Kshemkalyani. Causality-Based Predicate Detection across Space and Time. 54, 1438-1453. 2005. *IEEE Transactions on Computers*.

Robert Cooper & Keith Marzullo. Consistent Detection of Global Predicates. 163-173. 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

Vijay K.Garg & Brian Waldecker. Detection of weak unstable predicates in distributed programs. 5, 299-307. 1994. *IEEE Transactions on Parallel and Distributed Systems*.

Vijay K.Garg & Brian Waldecker. Detection of Strong Unstable Predicates in Distributed Programs. 7, 1323-1333. 1996. *IEEE Transactions on Parallel and Distributed Systems*.

Section Three: Methodology

Chapter 7 Methodology

7.1 Introduction

This research investigates the theoretical methods to validate CAN network based distributed automotive control systems. A validated system should satisfy the requirements from the users. The approach taken is to build a prototype global predicate evaluation system to check if the system under test validated. The approval is shown in Figure 7-1 . The prototype takes the data from the ECU network descriptions, evaluates the predicate according to these data and generates the validation result. In order to validate this result, a global predicate evaluation result should be generated manually. The result of the prototype evaluation should match the result that is evaluated manually; otherwise the prototype is not validated. This chapter will conceptually introduce the prototype.

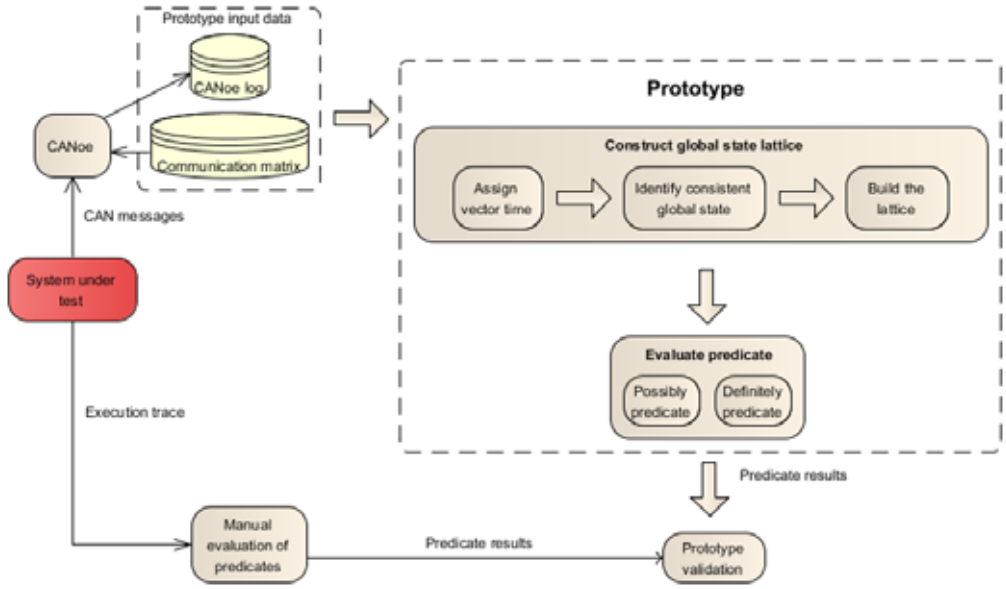


Figure 7-1 Global validation of distributed automotive control systems prototype

7.2 Construct global state lattice

The lattice that is evaluated by the predicate is constructed from the global states of the system. The global state is constructed from the local states of each node in the system. The global state has to be consistent otherwise it won't make any sense. For example, a message won't be received before it is sent; this never happens in the distributed system. As there is no physical global time in the system, to decide if the global state is consistent, the prototype needs to assign the logical time to each local state. The logical time used in the prototype is a vector clock. The whole procedure to build the lattice is shown in Figure 7-2 . Next sections are going to go through these three steps.

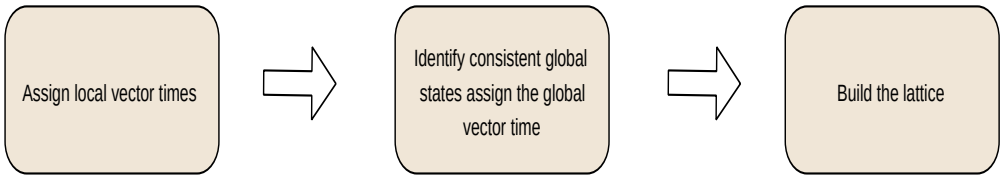


Figure 7-2 the procedure to build the lattice

7.2.1 Vector time assignment

The vector clock assignment depends on the local process event. When an internal event happens the node only increases its own clock. If it receives a message it updates all other node clocks from the sender's vector clock(Leslie Lamport 1978). The local state consists of node variables of interest in predicate evaluation. It can be logged by the XCP protocol.

The XCP measurement and the real network communication channel can be either on the same CAN channel or a different channel. The CANoe application records the whole network information. All messages on the network are synchronized by

CANoe real-time clock. The result is written into a log file, its format is shown in Table 7-1.

Real Time	CAN Channel	Message identifier	Message type	Frame type	Data length	Data

Table 7-1 CANoe log file format

Figure 7-3 is an example of the CANoe log file

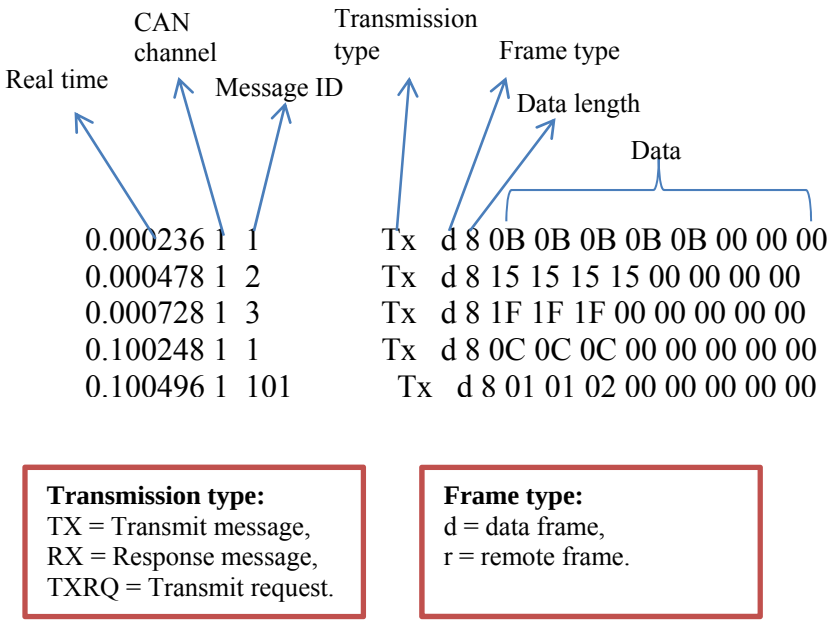


Figure 7-3 CANoe log example

The log file also has other different formats, but they won't be used in the prototype, so it is not necessary to introduce them.

There are two ways to record local state using XCP; one is time triggered that will send its local state continuously; another one is event triggered where the node only sends its state when the state changes. Each method has its own advantages and disadvantages; the comparison of them is shown in Table 7-2 .

	Communication traffic	Storage requirement	ECU application code modification
Time triggered	High	High	none
Event triggered	Low	Low	little

Table 7-2 time triggered and event triggered local state record mode

After the log file is generated, the local vector time needs to be assigned to the local state. There are two kinds of event that can cause the local state change: internal event and receiving event. An internal event is the event that only affects the its own local variable, and only updates its corresponding vector time component, e.g. timer triggered, panel button pressed, etc.. Receiving event occurs when the node receives a message. It causes the receiving node local variable change, as well as updating its corresponding vector time component. It also needs to update other vector time components based on the sender's vector clock. Therefore the sending event causes the other nodes vector time to change.

7.2.2 Identify consistent global state (CGS)

After all local states are assigned vector times; the next step is to construct the global state from these local states. An inconsistent global state is not meaningful in the sense that a distributed system can never be in an inconsistent state. To decide if the global state is consistent, the prototype needs to evaluate its local states' vector time; the algorithm to evaluate if the global state is consistent is (Ozalp Babaoglu & KeithMarzullo 1993) as follows:

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VT(e_i)[i] \geq VT(e_j)[i]$$

Equation 7-1 consistent cut

In the specification i, j represents the component (node) index in the vector time, n represents the total number of nodes, VT represents vector time, e represents event; it means if a global state is consistent, its local state vector time ith component of node i (local time) has to greater than or equal to local state vector time ith component of node j. Each node in the consistent global state has to have its latest corresponding vector clock component value. When the sending event happens, the receiving node knows the latest vector clock component of the sender, that's the situation $VT(e_i)[i] = VT(e_j)[i]$.

The following example demonstrates how to evaluate consistent global state. Figure 7-4 illustrates two processes' execution with vector time. Table 7-3 shows how to find consistent global state from the execution. The subscript number of e is the node number and the superscript number of e is the event counter (scalar time or local time). The numbers in the bracket is the vector time, first number is the scalar time of node 1 and second number is the scalar time of node 2. In the Table 7-3, the vector time of e_1^1 compares to the vector time of e_2^x; the red coloured number is the local time of node1 and the green coloured number is the local time of node 2. According to Equation 7-1 in the red numbers and the green numbers comparisons, both of them have to be true, for the global state to be consistent. The global state which is constructed by e_1^2 and e_2^3 is consistent in the case of $VT(e_i)[i] = VT(e_j)[i]$.

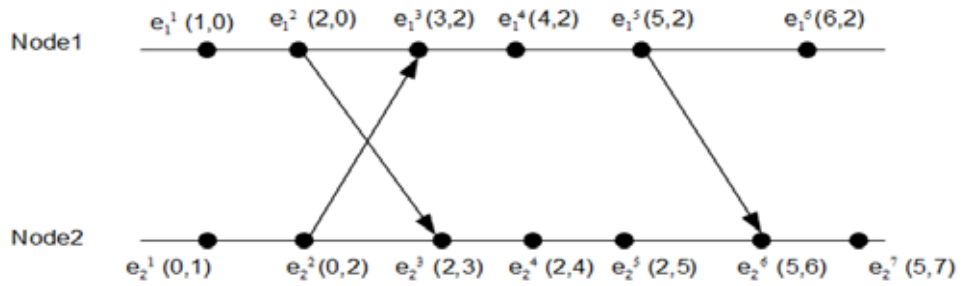


Figure 7-4 Two processes execution with vector time.

Node 1 event	**Node 2 event**	**Vector time comparison**	**result**
e_1^1	e_2^1	(1,0) (0,1) 1>0 and 0<1	Consistent
e_1^1	e_2^2	(1,0) (0,2) 1>0 and 0<2	Consistent
e_1^1	e_2^3	(1,0) (2,3) 1>2 and 0<3	Inconsistent

Table 7-3 evaluate CGS example.

The example only shows a two nodes network, for three or more nodes the system need to compare each node to all the other nodes.

If the global state is consistent, then the global vector time can be assigned to this CGS. Each component of the global vector time is the highest value of each node's corresponding vector time component, e.g. for the CGS $\{e_1^1, e_2^1\}$, its global vector time is (1,1) .

7.2.3 Build the lattice

After all consistent global states are built; the next step is to build the lattice. Figure 7-5 is an example of a two node execution lattice. The level on the right of the lattice is defined as the sum of the global vector time (Keith Marzullo & Gil Neiger 1991). In the same level, it is possible to have different CGS, it means at the same time the system can be in any one of the same level consistent global states; depending on the actual execution run.

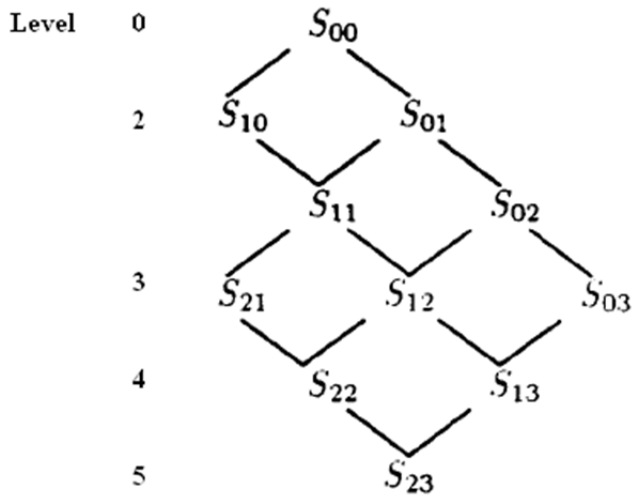


Figure 7-5 Two node execution lattice example

The execution goes through levels; each CGS may have a corresponding reachable CGS, it has to increase one corresponding vector time component by one to reach the next CGS, e.g. in Figure 7-5 from S_{02} to S_{12} and S_{02} to S_{03} is reachable, but it is not reachable from S_{02} to S_{21}.

From the example lattice it can see that one application can go through different consistent global state sequences. It is non-deterministic, so the next step needs to evaluate if it goes through expected or unexpected states by specifying a predicate.

7.3 Evaluate predicate

For a given predicate (Φ) there are two modalities: Possibly predicate and Definitely predicate. They are defined as (Robert Cooper & Keith Marzullo 1991)

Possibly predicate: "For all executions consistent with the observed behaviour, there was some point in real time at which the global state of the system satisfied Φ**.**"

Definitely predicate: "For *all* executions of *P* consistent with its observed behaviour, Φ was true at some point.

Predicates are evaluated on the local states of nodes.

The prototype system should be able to take any combination of ECUs' values to evaluate if it is definitely or possibly true.

7.4 Validation tests

After the prototype is built, it is necessary to validate it. The method to validate the prototype program is

1. Specify a predicate
2. Evaluates the test when system execution lattice by manually checking if the global states in lattice satisfies the specified predicate.
3. Evaluate the under test system execution lattice by prototype.
4. Compare the manually generated result against the prototype result; if they are match the prototype is validated, otherwise it is invalidated.

7.5 Conclusion

The common problem for distributed system debugging is no physical global time and shared memory exists. Some research on solutions has been done, but they are mostly for the Ethernet (Kenneth P.Birman 1995, p288-292), not for the distributed automotive system.

For the moment, the most reasonable method to debug the distributed automotive system is to evaluate the predicate of the system execution lattice, which is constructed by the consistent global states. A consistent global state is built using the local state of each node, the selection of the local state is based on the causality or logical time (vector time).

This research will build prototype software based on the GPD method to validate CAN network based distributed automotive control system. The effectiveness of the prototype in achieving system validation goals will be evaluated.

References

Keith Marzullo & Gil Neiger. Detection of Global State Predicates. [LNCS 579], 254-272. 1991. Proceedings of the 5th Workshop on Distributed Algorithms.

Kenneth P.Birman. Building Secure and Reliable Network Applications. 1995. Department of Computer Science Cornell University Ithaca, New York 14853.

Leslie Lamport. Time clocks and the ordering of events in a distributed system. 558-564. 1978. Communications of the ACM.

Ozalp Babaoglu & KeithMarzullo . Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. 1993. Italy, Laboratory for computer science university of bologna.

Robert Cooper & Keith Marzullo. Consistent Detection of Global Predicates. 163-173. 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

Section Four: Implementation and Testing

Chapter 8 Prototype Development

8.1 Introduction

The methodology to validate the CAN network based distributed automotive control system has been given in the previous chapter. This chapter will describe the design of the prototype software.

The first section gives an overview of the design of the prototype. The second section describes the data requirement for the prototype. The third section talks about how to generate the test cases for the prototype. The fourth section is the detail of the prototype design.

8.1.1 Design overview

There are two main parts for the prototype software design.

1. The test case generation system uses UML to interpret the structure of the test case and uses software to generate the test case described by the UML.
2. The prototype software evaluates execution lattice of CAN network ECUs.

8.1.1.1 Test case generation system

For the purpose of conveniently validating the prototype; the test cases can be generated as CAPL code that simulates the ECUs. The CAPL code acts as a state machine, any internal or receiving events happening will cause a state change. The overall process of generating the test case can be illustrated as Figure 8-1

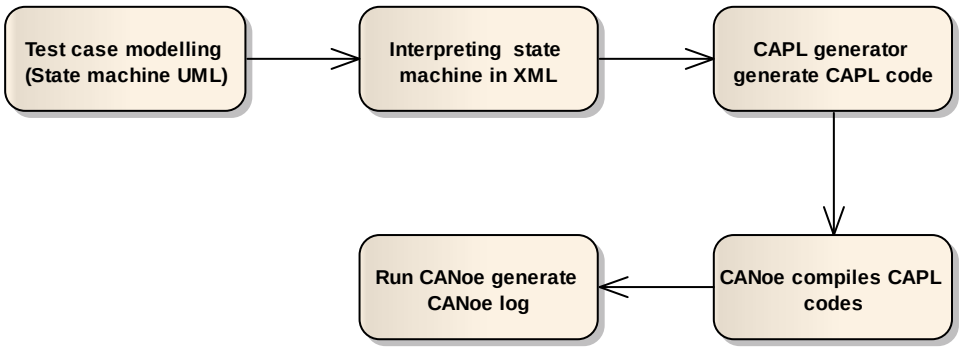


Figure 8-1 test case generating progress

The sequence for test case generation is

1. Using UML to model the test case which is described in a state machine diagram.
2. Using a state machine template creates the XML file for each node.
3. Using CANoe to compile these CAPL codes.
4. Running the CAPL code, using CANoe to log messages passed on the CAN network.

8.1.1.2 Prototype software

Based on the methodology that has been identified in Chapter 7, the prototype structure can be generally illustrated as Figure 8-2.

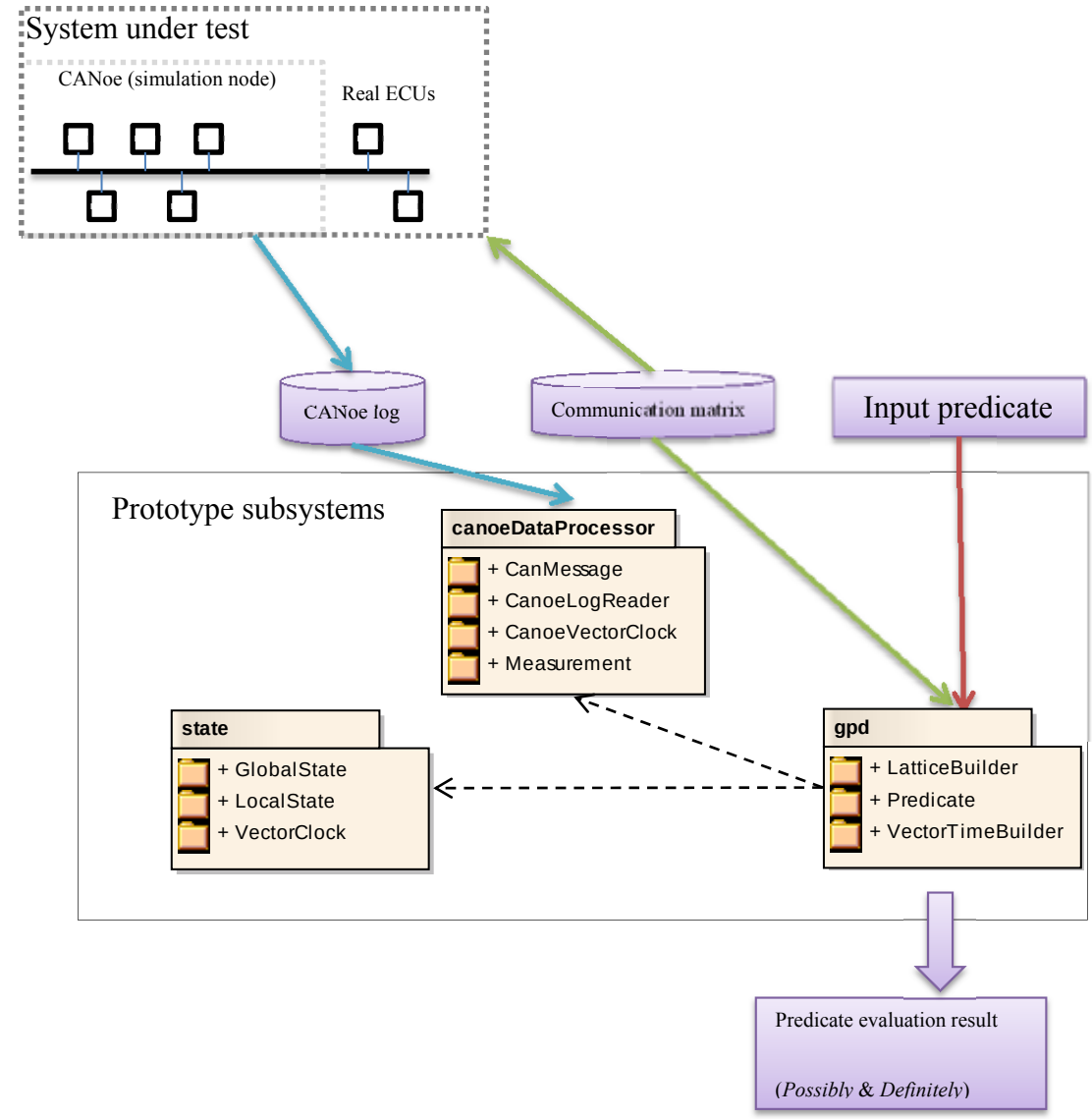


Figure 8-2 prototype design overview

The diagram shows: CANoe records the CAN messages and environment variables of the system under test to the CANoe log file. The node of system under test can be real ECUs and/or simulated ECUs using CAPL code. To run the simulation also needs to create a CANdb database that describes the bus data in symbolic terms. CANdb contains the communication matrix that describes the senders and receivers of all messages. After the CANoe log is generated, the prototype subsystem *canoeDataProcessor* reads this log file and encapsulates log

data into classes. The subsystem *state* is used as data structures to encapsulate the state element. The subsystem *gpd* assembles the encapsulated measurement data from CANoe log and associated vector time as a local state. It evaluates assembled local states to construct the consistent global states and assigns global vector times to these consistent global states. According to the global vector time of the consistent global state *gpd* builds the system execution lattice. For a given *predicate*, *gpd* can evaluate if it is a *definitely* or a *possibly predicate*. The *gpd* package is the core package of the whole prototype. Most of the processing is done by *gpd*. There are also other packages for convenient operation of the prototype and representing the result. They will be introduced in a later section.

The next sections will give the detail of how the prototype is designed, but first of all it is necessary to introduce the data required for the prototype.

8.2 Implementation tools

For the prototype design, the following tools are used.

- Enterprise architecture (EA): modelling the state machines of the node.
- Visual studio: CAPL code generator (C#).
- CANoe: it is a comprehensive software tool for the development, testing and analysis of entire ECU networks and individual ECUs. It runs CAPL code, monitors CAN bus, generates CANoe log file.
- Eclipse: all predicate evaluation programs are coded in java. Eclipse offers a powerful Integrated Development Environment (IDE) for Java.

8.3 Data requirements

There are four types of file required for the test case and the prototype. Data used by test case generation are UML (test case modelling), XML (test node template). Data used by prototype are asc (CANoe log file), XML (communication matrix). They will be introduced in this section one by one.

8.3.1 UML test case modelling

The UML diagram is used to model the test case with state machine diagrams; it gives a clear conceptual view of the test case. The test case is constructed as state machines, because the real ECU works similar to state machine as well. Figure 8-3 is an example of a simple state machine; there are three states in this state machine, the state1 sets timer *t_n1_1* as 50 milliseconds, when the timer *t_n1_1* expires the transition happen, the state machine will be in the state2; in state2, it sends message *msg_n1_1* and set *t_n1_2*. When the timer *t_n1_2* expires, the transition happens, the state machine goes to state3; there is no action in state3; when the state machine receives message *msg_n2_1* (message ID is hex 102), the transition makes the state machine go to state1; the state machine will repeat the process.

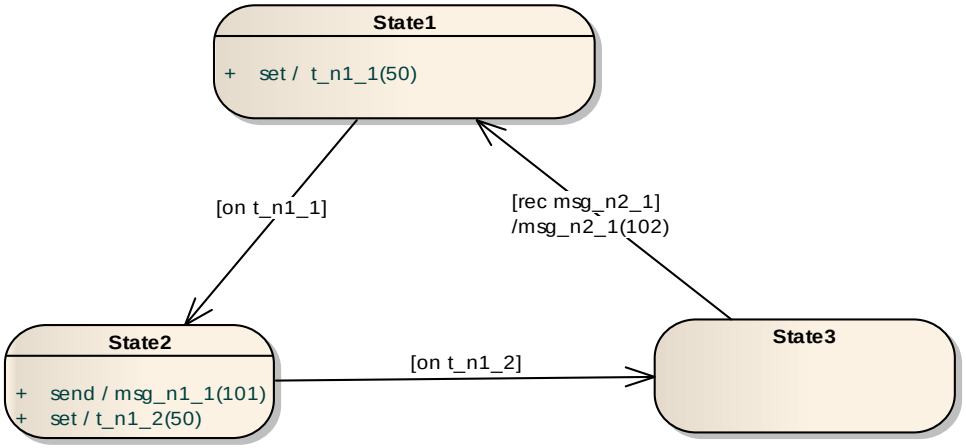


Figure 8-3 state machine example

In any state, the state machine can set timer and/or send message; the timer name is as *t_n1_1*, the *n1* means node *1*, the following *1* means state *1*. The message name is similar to the timer name, the only different is the message variable starts with *msg*. The transitions can be caused by timer expiring, receiving message, and environment variable changes (the variable on the CANoe panel changes. e.g. button pressed etc.). The transitions timer expiring and environment variable changing uses “on” before the timer variable and environment variable; The transition receiving message uses “rec” before the message variable.

Based on Figure 8-3, the corresponding CAPL code is shown in Figure 8-4. CAPL program manual is (Vector CANtech 2004). To use CANoe/CAPL to simulate test case nodes is quicker than programming individual ECUs.

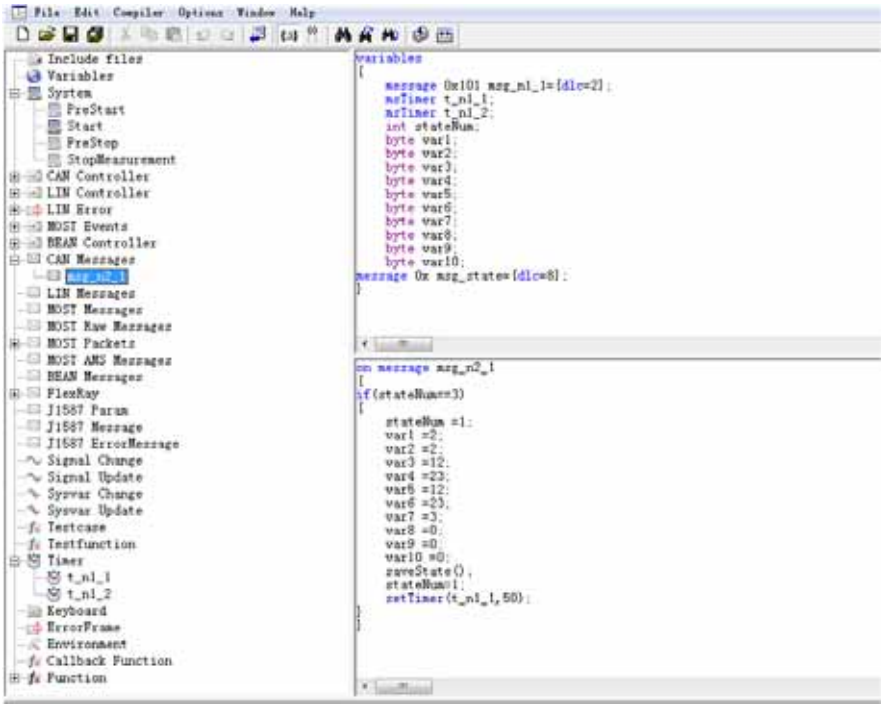


Figure 8-4 CAPL code

8.3.2 CAPL code generator XML schema

The UML specification of the state machine node can be manually saved in XML format. This XML file is used as a template to generate the CAPL code which simulates the ECU on the CAN network. The XML file structure can be described by the XML schema. The state machine XML template will be constructed by using this schema. A state machine node can be described as Figure 8-5 .

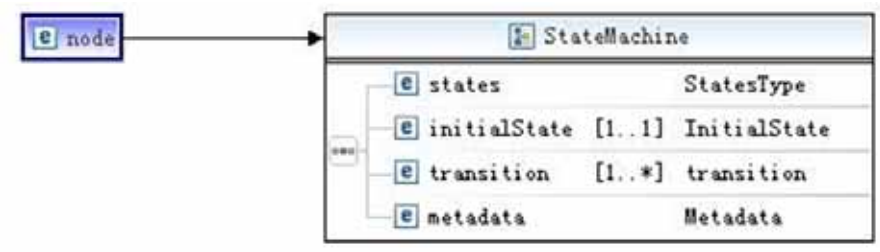


Figure 8-5 state machine node (from Eclipse UML2.1 plug-in)

There are four components in a state machine node; *states*, *initialState*, *transition*, *metadata*. They are instances of four different types that is illustrated in Figure 8-2

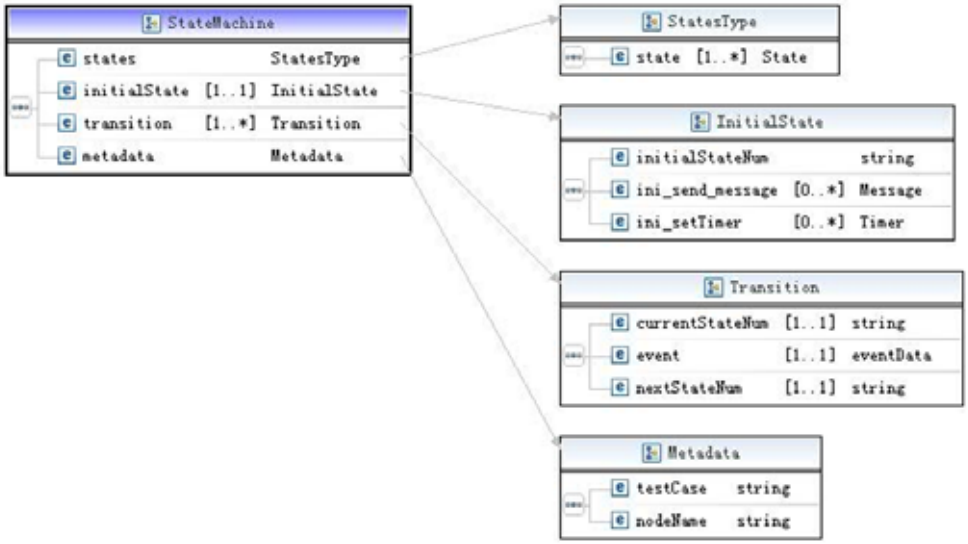


Figure 8-6 state machine node component types

These four types also contain subtypes; the following subsections will individually describes all types of data used by the state machine schema.

8.3.2.1 StatesType

StatesType consists of one or more State type elements. A state type element is a local state of the state machine node. Each state type has its state number (state index) and variables (local states). The *StatesType* structure illustrates as Figure 8-7

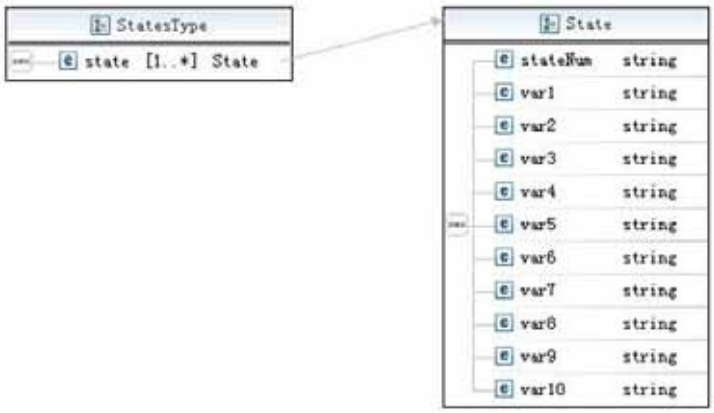


Figure 8-7 StatesType

8.3.2.2 InitialState

InitialState type element describes the initial state and event of the state machine. It is illustrated in Figure 8-8

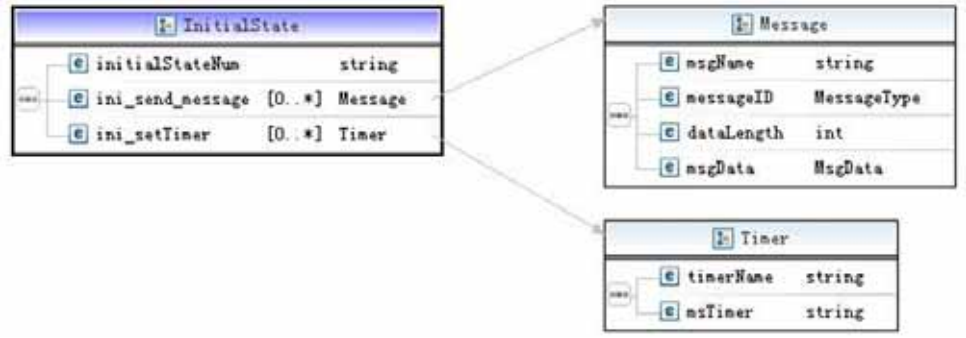


Figure 8-8 Initial state type

The *initialStateNum* is the initial state number (state index); the events that *possibly* happen in the initial state are send message and set timer. *ini_send_message* and *ini_setTimer* is instances of Message type and Timer type.

The structure of Message type is similar to a CAN message. *msgName* records the name of the message. *messageId* is type of *MessageType* that is a selection type: it can either be a standard message which uses standard CAN message ID or an extended message which uses extended message ID. The *MessageType* type is illustrated in Figure 8-9 ; *dataLength* describe the length of the data frame. *msgData* records the message data that can contain 0 to 8 bytes, *MsgData* type is illustrated in Figure 8-9

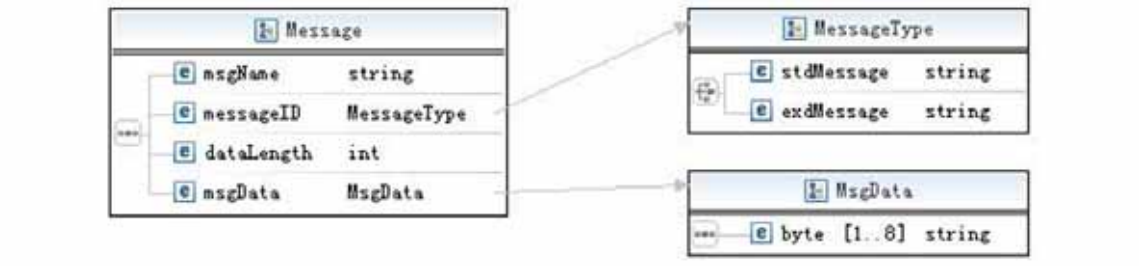


Figure 8-9 Message type

The structure of Timer type records the name of the timer (*timerName*) and the duration of the timer (*msTimer*); it is illustrated in Figure 8-10

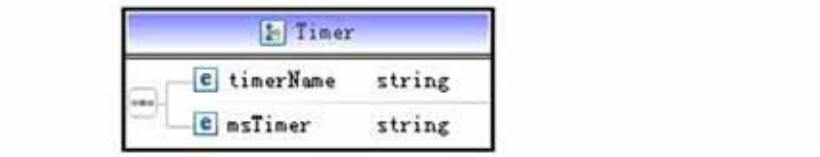


Figure 8-10 Timer type

8.3.2.3 Transition

The transition type data records current state number, the event that causes the state change and next state number. Figure 8-11 shows the structure of the Transition type. Variable *currentStatNum* and *nextStateNum* is the number of a state and event is an instance of *eventData* type.

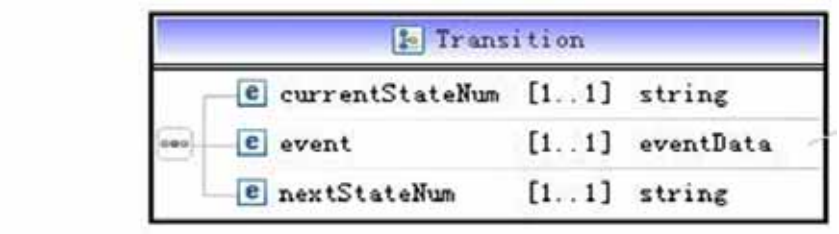


Figure 8-11 Transition type

eventData type consists of *eventTag*, action, *send_message*, *setTimer*. It is illustrated in Figure 8-12. *eventTag* is instance of *ActionType* that is a selection type, it records event tag that is used as boundary in the CAPL code to separate different events, the *ActionType* is illustrated in Figure 8-12. Action type records the variable that is triggered by event, it is used as handler to handle the event e.g. a timer variable t_n1_1 is set to 100 milliseconds, when t_n1_1 expires, the handler "on t_n1_1" will handle the timer expiry event. In the handler, the local states are changed and the messages can be sent and/or the timers can be set as well. The Message and Timer types have been introduced in 8.3.2.2 .

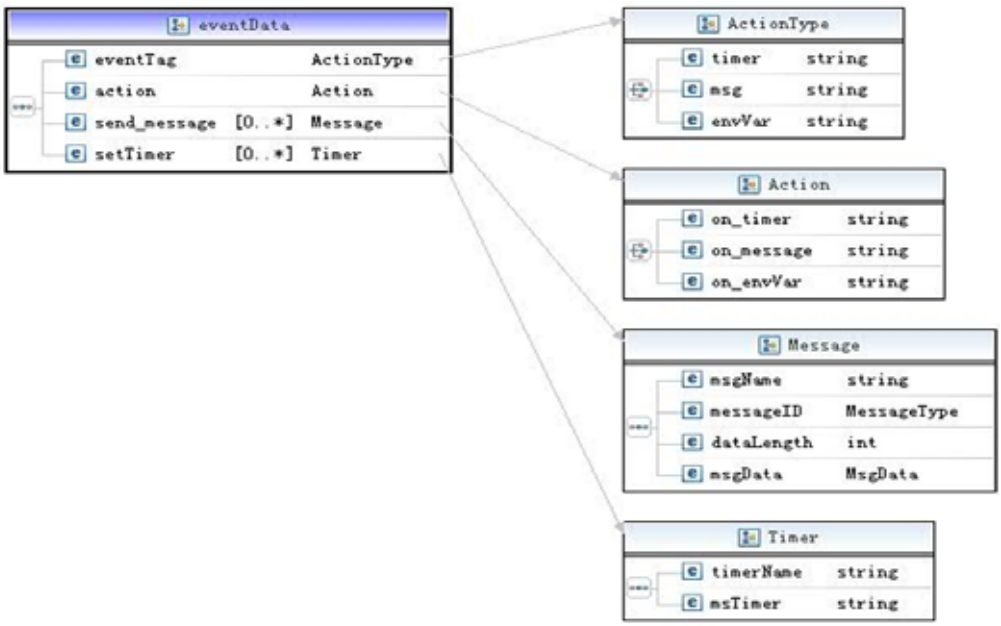


Figure 8-12 *eventData* type

8.3.2.4 Metadata

Metadata is used to record the information about the test case and the node name in the case. It is illustrated in Figure 8-13

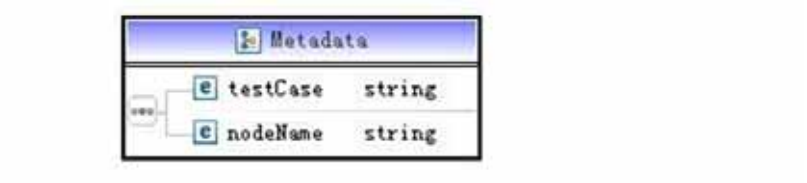


Figure 8-13 Metadata type

8.3.3 CANoe log file

The CANoe log file is generated by CANoe. It logs the information about transmitted CAN messages and environment variables. The structure of the file has been described in section 7.2.1.

The local state of ECUs (simulated and/or real) will be recorded by the CCP (CAN Calibration Protocol) either on the same or different network. CCP is also CAN messages, thereby the CANoe logs the local states of each nodes in a single CANoe log file. The CANoe timestamp will be used to order local ECU state and inter ECU messages. The local states can be stored by CCP either continually or triggered by the state change. For continually log the local states, the frequency of CCP data logging is important to ensure that all internal events changes are detected.

8.3.4 Communication matrix

Communication matrix describes the information about the sender and receivers of the corresponding CAN message ID. The CAN message ID has to be unique on the same CAN network: only one node can send a specific ID message, but it can be received by different nodes.

The communication matrix is contained in CANdb, but it is in the different format that is not convenient to use by prototype program. For the prototype system, it is manually migrated to XML format.

The communication matrix schema contains zero or more message type elements; a message type element contains the attributes of the message id and the number of the sender node, it also contains a list of the receiving node numbers. The structure of the communication matrix illustrates in Figure 8-14

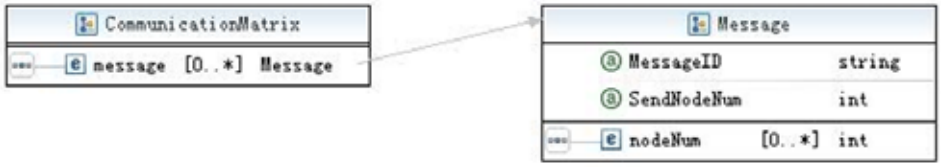


Figure 8-14 Communication Matrix structure

All the data required by test cases and the prototype has been introduced in this section. The next section will describe the design of the test cases and the prototype.

8.4 Test case program design

8.4.1 CAPL code generator

CAPL code generator is used to convert the XML state machine node description to CAPL code. It is programed in C#. There are two main classes of the CAPL code generator: *Form* and *CodeGenerator*.

8.4.1.1 Form class

The Form's class diagram is shown in Figure 8-15, it has two main event operations; CAPL code XML template selection button *selXML_btn_click* gives a file selection dialog to let users select the XML template file. CAPL code generation button *generateBtn_Click* opens a saving path dialog to let the user choose the saving path of the CAPL code.

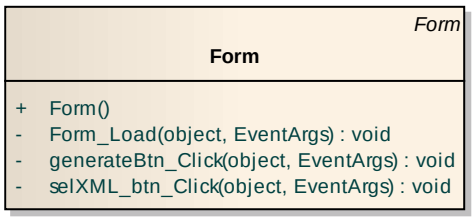


Figure 8-15 Form class diagram

Next is an example how to use the GUI. The GUI of the code generator is shown in Figure 8-16

Figure 8-16 CAPL code generator GUI

The select XML button lets the user select the state machine XML template as shown in Figure 8-17

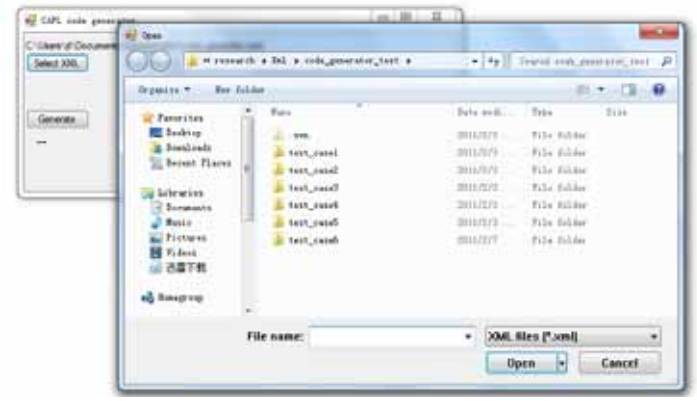


Figure 8-17 CAPL code generator select XML template dialog

After the XML template file is selected, click the *Generate* button to choose the desired path and give the name of the CAPL code file to be saved. This is shown in Figure 8-18 .

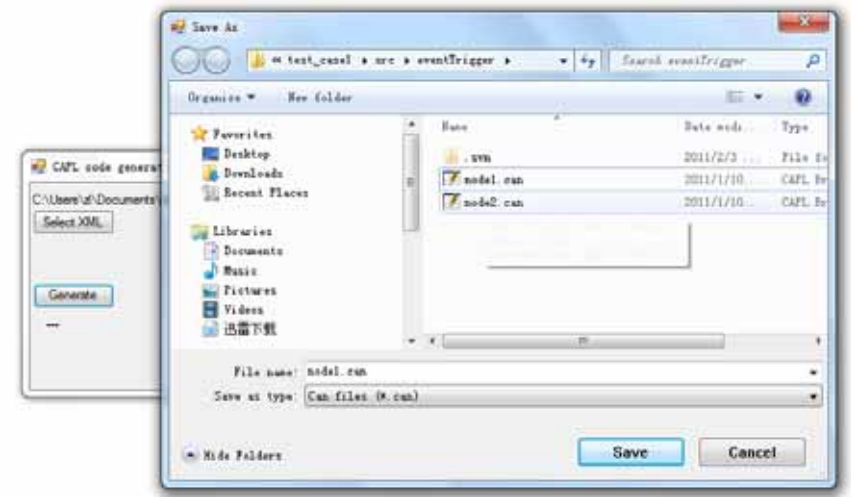


Figure 8-18 saving CAPL code dialog

8.4.1.2 CodeGenerator class

The *CodeGenerator* class does the actual job of converting the XML template to the CAPL code. Figure 8-2 is the class diagram of the CAPL code generator. It is more like procedure program; it takes the input CAPL code template XML file and generates the CAPL code. The red texts in the diagram are the variables of the

class, and the green texts are the functions of the class. The main function in the program is *generateCode*. It takes as parameter the string of the XML template path to generate the CAPL code.

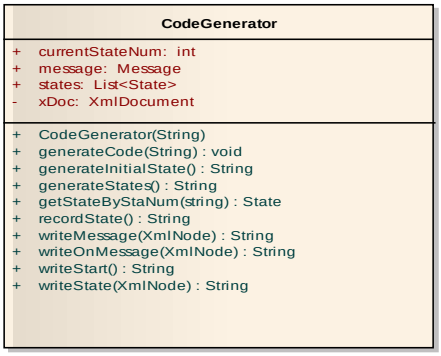


Figure 8-19 CAPL code generator class diagram

The main procedure in the *generateCode* function is illustrated in Figure 8-20. It starts with generating the initial state code that includes the code to define the variables, initial state, and initial actions for the state machine node. Next activity is a loop to generate the rest of the states codes; the final activity writes the CAPL function code to record the state. There are also subordinate activities in the "generate initial state" and "generate state" activities.

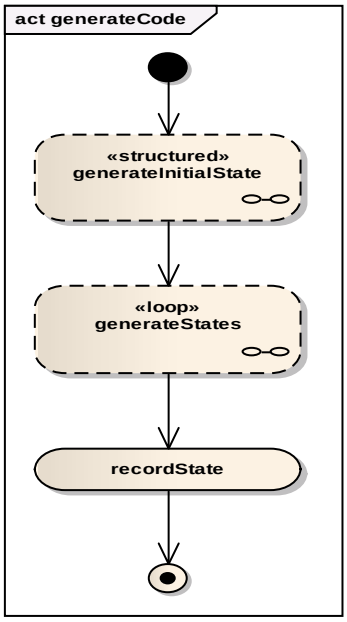


Figure 8-20 CAPL code generator main procedure

The activities procedure to *generateInitialState* is illustrated in Figure 8-21 . The activity *generate variable* writes the variable declaration for the CAPL code; the activity *writeStart* codes the initial state and actions.

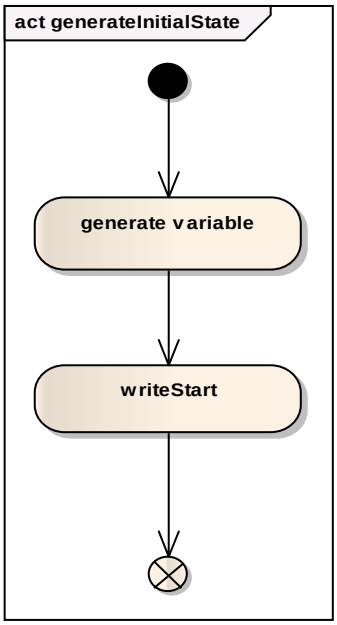


Figure 8-21 activities procedure generate initial state

In order to generate state, the *generateStates* activity contains a loop to call the *writeState* function. The *writeState* function codes the state and transition events of the state machine. The activities procedure in the *writeState* fucntion is shown in Figure 8-22. The procedure updates states first, after checking for any send message and/or set timer events. If any of them are recorded in the XML template, they will be coded in the CAPL code.

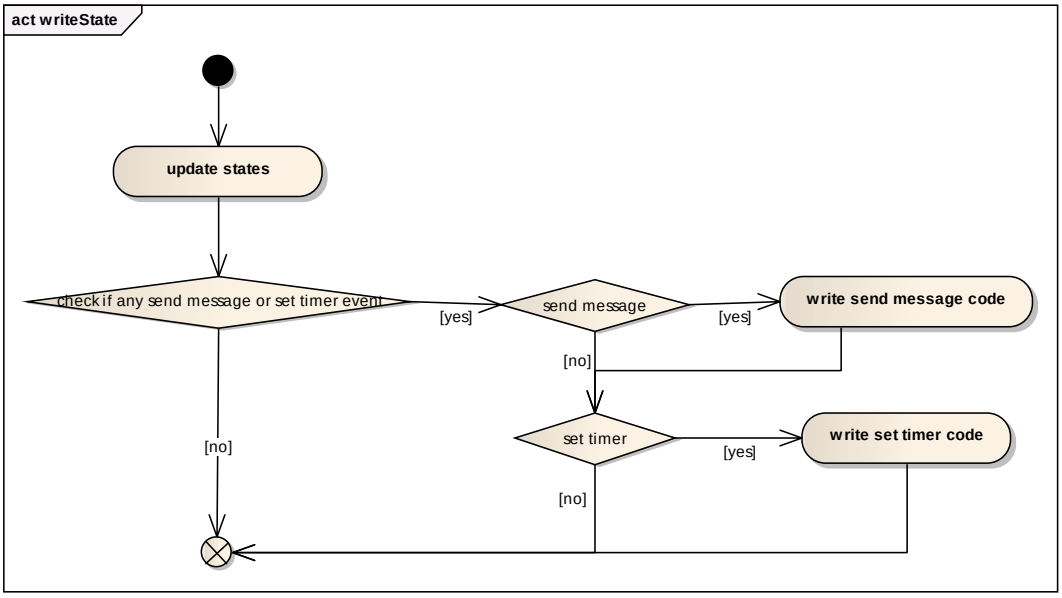


Figure 8-22 activities procedure in the *writeState* function

After all CAPL nodes codes are generated, they are compiled and run by CANoe. CANoe generates the CANoe log file for the later processing. The following sections will describe the design of the GPD prototype.

8.5 GPD prototype program design

This section introduces the design of the prototype. There are three main packages (*canoeDataProcessor*, *state*, and *gpd*) to implement the predicate evaluation as

shown in Figure 8-2. Each package contains classes. These classes collaborate together to achieve the predicate evaluation result. Figure 8-23 is the class diagram to describe the relationship between the classes.

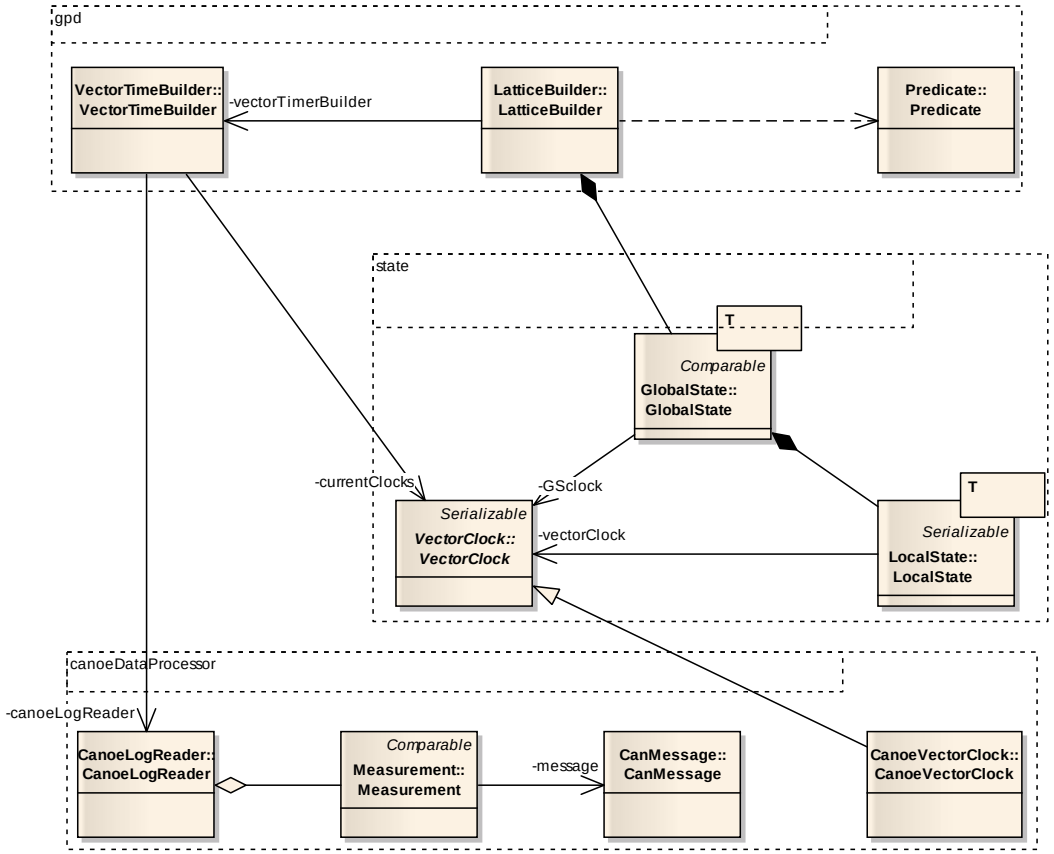


Figure 8-23 Prototype class diagram (only main classes)

In the class diagram, the packages are separated by the dashed rectangle. The following sections describe these packages.

8.5.1 *canoeDataProcessor* package

canoeDataProcessor package is used to read CANoe log file and encapsulate CANoe log data. There are four classes in this package. Their relationships are shown in Figure 8-24.

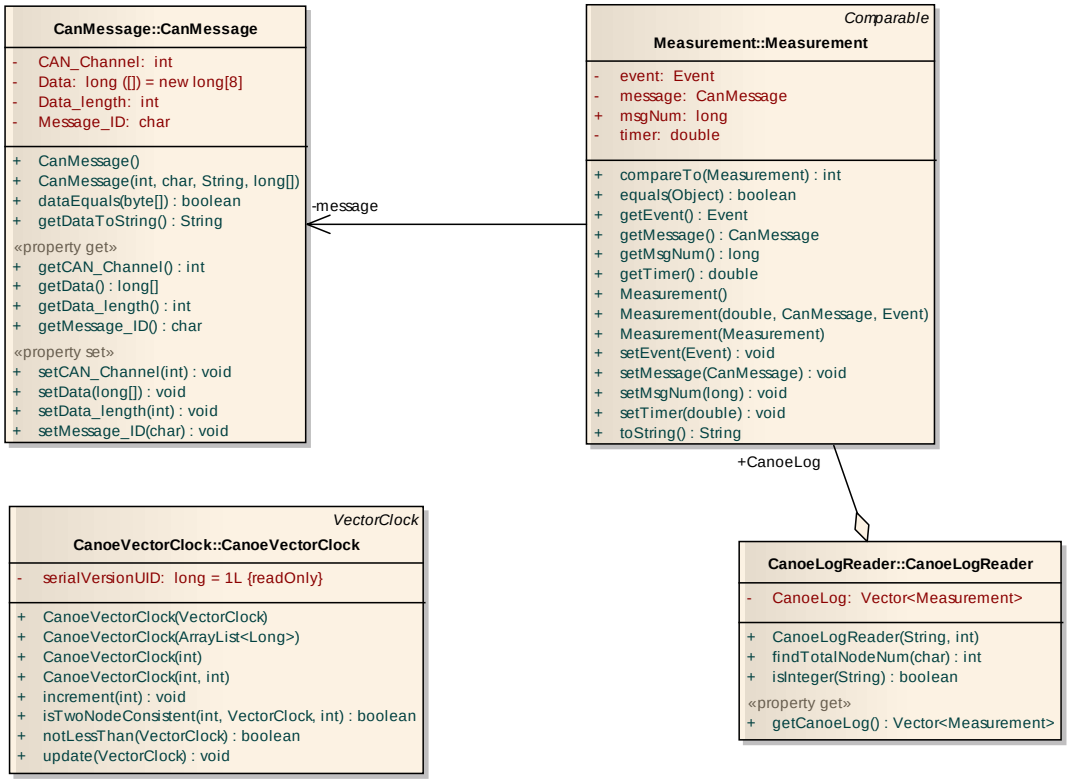


Figure 8-24 class diagram of *canoeDataProcessor* package

8.5.1.1 CanoeVectorClock class

The *CanoeVectorClock* class is a child class of *VectorClock* that is an abstract class contained in *state* package. It specializes the *VectorClock* class. The *CanoeVectorClock* class inherits the attribute *vectorClock* that is *Long* type of *ArrayList* from the *VectorClock* class. The attribute *vectorClock* is the data structure to store the vector clock. A *CanoeVectorClock* has operations to increase a given component of its own *vectorClock*, compare its own *vectorClock* with another *vectorClock*, update its own *vectorClock* from another *vectorClock*, and check if its own *vectorClock* is consistent with another *vectorClock*. All attributes and functions of the *CanoeVectorClock* class are shown in Figure 8-24.

8.5.1.2 CanMessage class

The *CanMessage* class encapsulates the CAN message. It has attributes: *CAN_Channel*, *Message_ID*, *Data_length*, and *Data*. *CAN_Channel* defines the CAN bus channel used. *Message_ID* defines message ID. *Data_length* defines the length of the data. *Data* defines the message data.

8.5.1.3 Measurement class

The *Measurement* class encapsulates the item in the CANoe log. A CANoe log item contains the real time of the message sent and the CAN message frame (described in Chapter 7). The *Measurement* class contains attribute of the *CanMessage* type and attribute of the *Double* type (real time). The CANoe log item stores the node state in the format of the CAN frame. But it does not have any knowledge about the type of the event that can be found by the other functions. So it is necessary to have an attribute of Event type. The Event is an *enum* type, it has three elements: *receiveMsg*, *sendMsg*, and *internalEvent*. They are the three possible events: receive message, send message, and internal event.

8.5.1.4 CanoeLogReader class

The *CanoeLogReader* class reads the CANoe log to extract the data. Depending on these data, it creates the *Measurement* objects and stores these objects into a *Vector*. The only attribute in the *CanoeLogReader* class is the *Measurement Vector* called *CanoeLog*.

8.5.2 *state* package

The *state* package defines the elements and the structures of the states (global and local). There are three classes in the *state* package: *VectorClock*, *LocalState*, and *GlobalState*. Their relationships are illustrated in Figure 8-25.

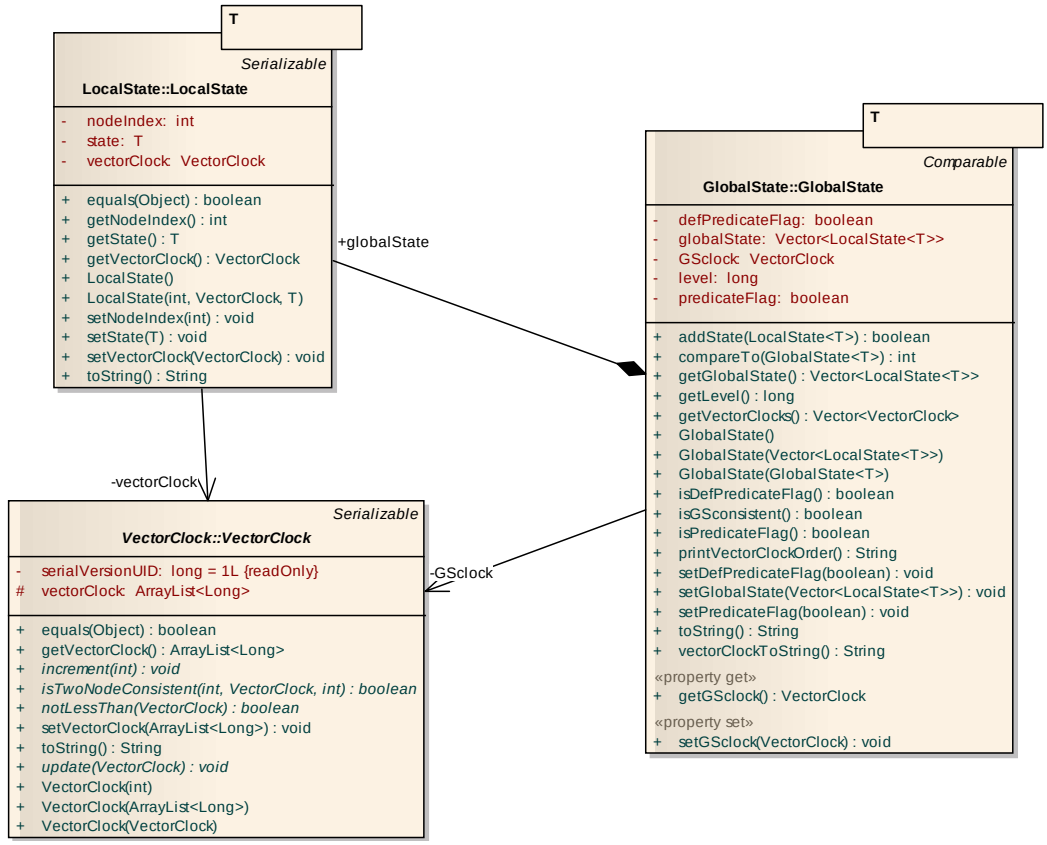


Figure 8-25 class diagram of *state* package

8.5.2.1 VectorClock class

The *VectorClock* class is an abstract class. It generalizes the type of vector clock. The *VectorClock* class defines the structure and the behaviors of the vector time. It is specified by the *CanoeVectorClock* class. More detail can be found in section 8.5.1.1 .

8.5.2.2 LocalState class

The *LocalState* class defines the local node structure. It is a generic class, so it can hold objects of any state class. For the prototype, the state class used is the *Measurement* class. A *LocalState* class contains three attributes; *nodeIndex*, *state*, and *vectorClock*. The *nodeIndex* is an integer. It records the index of the node.

The *nodeIndex* minus 1 is its corresponding vector clock component index. The attribute *state* can is a generic type, it holds the state value of the node.

8.5.2.3 GlobalState class

The global state is the collection of the local states, and has its own global vector clock. The *GlobalState* class has the attributes of *LocalState Vector* type and *VectorClock* type. For the purpose of detecting the *possibly predicate*, the attribute *predicateFlag* is used as a flag to indicate if the global state satisfies the predicate. Also another attribute *defPredicateFlag* is used to find *definitely* predicate. It indicates the global state that does not satisfy the predicate is reachable from the initial state. If the unsatisfied predicate global state is reachable, then the value of *defPredicateFlag* of all global states in the path has to be false. The attribute *level* is used to present which level the global state belonged to. There is a function called *isGSconsistent*. It is used to check if the global state is consistent. *isGSconsistent* implements the algorithm in section 7.2.2.

8.5.3 *gpd* package

The gpd package is the core package of the prototype. All logical GPD processes are implemented by the classes in this package. There are three classes in *gpd* package; *Predicate* class, *VectorTimeBuilder* class, and *LatticeBuilder* class. Their relationships are illustrated in Figure 8-26.

Figure 8-26 class diagram of *gpd* package

8.5.3.1 Predicate class

The *Predicate* class is a structure to hold a *predicate*. A *predicate* consists of a state variable, relational operator, constraint value, and logical connective which can connect to another *predicate* (e.g. "||" and "&&"). The corresponding attributes of the *Predicate* class are *valueIndex* which is the index of the local state array, *operator*, *predicate*, and *relation*. The attribute *nodeNumber* indicates the index of the node. An example of a *predicate* is "node[1].stateArray[0]>21 ||" .

8.5.3.2 VectorTimeBuilder class

The *VectorTimeBuilder* class assigns the vector time to the local state. It uses attribute *canoeLogReader* to read the CANoe log file, stores the data of the CANoe log into a *Measurement Vector* (the attribute *canoeLog*), loads communication matrix data, and goes through the *Measurement Vector* to assign the value to the instance of the *LocalState*. The order of assigning the vector clock is illustrated in Figure 8-27.

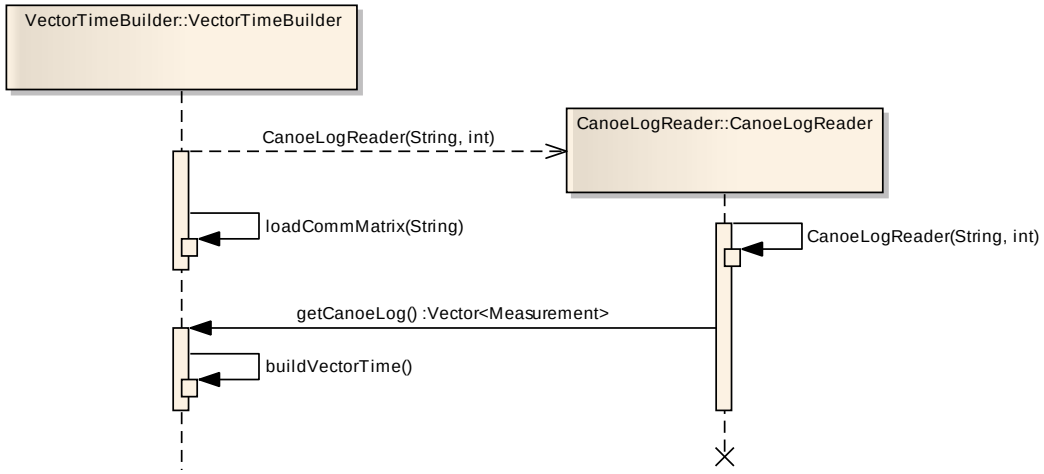


Figure 8-27 sequence diagram to assign vector time

A *HashMap* type data structure (the attribute *nodesLocalStateMap*) is used to store the local states of different nodes. The *key* of the *nodesLocalStateMap* is the index of the node, and the *value* of the *nodesLocalStateMap* is a *LocalState Vector*. Each node has its corresponding list of local states. Another *HashMap* type data structure (the attribute *commMatrix*) is used to store the data of the communication matrix. The *key* of the *commMatrix* is the CAN message ID. The value of the *commMatrix* is an *Integer Vector*. The first element of the vector is the index of the sending node; the other elements are the indexes of the receiving

nodes. Because the real system communication and CCP uses the same CAN channel, the attribute *threshold* works as a separator to separate them. If the CAN message ID is smaller than *threshold*, then the message is a CCP message. Otherwise it is a real system communication message.

The process to build the vector time from the *Measurement Vector* is quite complex. The whole process is done by *buildVectorTime()* function as shown in Figure 8-27. The working flow of the function *buildVectorTime()* is illustrated in Figure 8-28.

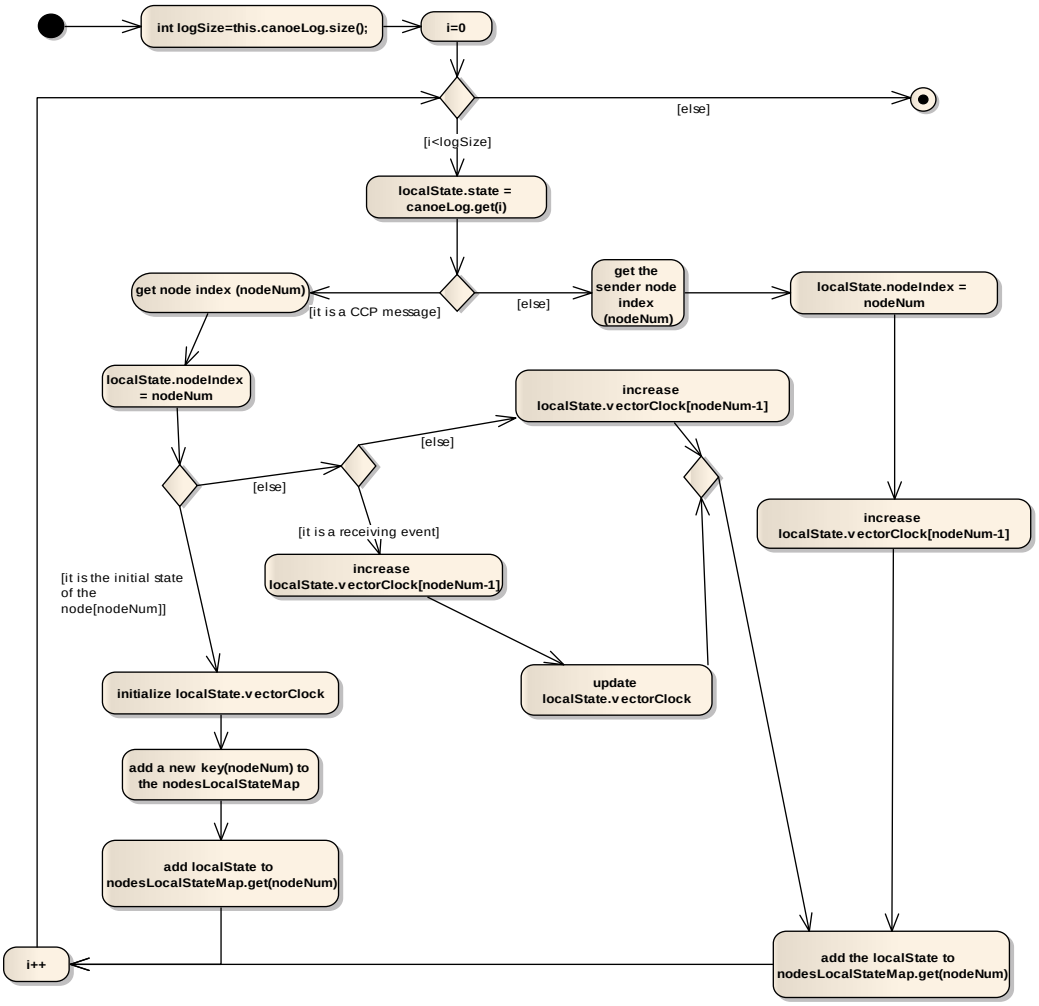


Figure 8-28 working flow of the function *buildVectorTime()*

As shown in the activity diagram (Figure 8-28), the *buildVectorTime()* function is based on a big for-loop. The variable i is the counter of the loop. Each time through the loop a new *LocalState* object called *localState* is created. The variable *localState* does not exist in the real code. It is only used for explanation purposes and makes it easier to understand the diagram. At the start of the loop iteration the state of the *localState* is assigned to the i^{th} *canoeLog* element. The loop goes through the *canoeLog* evaluating each measurement, to find which node local state it belongs to and to assign the vector time to the local state. There are a few condition statements to implement such evaluation. The first condition statement is to separate the CCP message and the real system communication message. They are separated by the *threshold*. If it is a real system communication message, then the sending node index (*nodeNum*) will be found by searching the communication matrix; the *nodeNum* is assigned to the variable *state* of the *localState*; the $(nodeNum - 1)^{th}$ vector clock of the *localState* increases one; the *localState* is added to the *LocalState Vector* in the *nodesLocalStateMap* mapped by the key which is the *nodeNum*. If the message is a CCP message, then it will meet the second condition statement. But before the second condition statement, the *nodeNum* is found by the message ID (the message Id is the node index). The second condition statement separates the initial state and the non-initial state. To judge if the *localState* is the initial state of a node, it needs to check if the value of the *nodeNum* mapped in *nodesLocalStateMap* is null. If null it is an initial state, otherwise it is a non-initial state. If it is an initial state, then the following process will be pretty simple. The process is initializing the vector clock of the *localState*, adding the new key *nodeNum* to the *nodesLocalStateMap* and adding the first element to this key mapped *LocalState Vector*. If the *localState* is non-initial state,

then it goes to the third condition statement. The third condition statement separates the event types which causes the state change. Here the event types are internal and receiving events, the sending event has been filtered out by the first condition statement. If the *localState* is caused by an internal event, then $(nodeNum - 1)^{th}$ vector clock of the *localState* increases by one; the *localState* is added to the *LocalState Vector* in the *nodesLocalStateMap* mapped by the *nodeNum*. If the *localState* is caused by a receiving event, the process is similar to the internal event. The only difference is after the vector clock increased, it needs to update the vector clock of the *localState* from the vector clock of the sender. The index of the sender can be found by searching the communication matrix. The vector clock of the last state of the node pointed by this index is the vector clock that the *localState* should update from.

8.5.3.3 LatticeBuilder class

The *LatticeBuilder* class evaluates the consistent global states, constructs lattice and detects the predicate. It consists of two attributes: *vectorTimeBuilder* and *lattice*. The type of the *vectorTimeBuilder* is *VectorTimeBuilder*. The type of the *lattice* is a HashMap, the type of the key of the HashMap is *Integer*. The key indicates the level of the lattice. The value type of the HashMap is *GenericTreeNode Vector*. The *GenericTreeNode* class is a tree type of the data structure. In the lattice, a node may have child nodes and/or parent nodes. A *GenericTreeNode* holds a global state, and also maps the child and parent relationships. The structure of the *lattice* is illustrated in Figure 8-29.

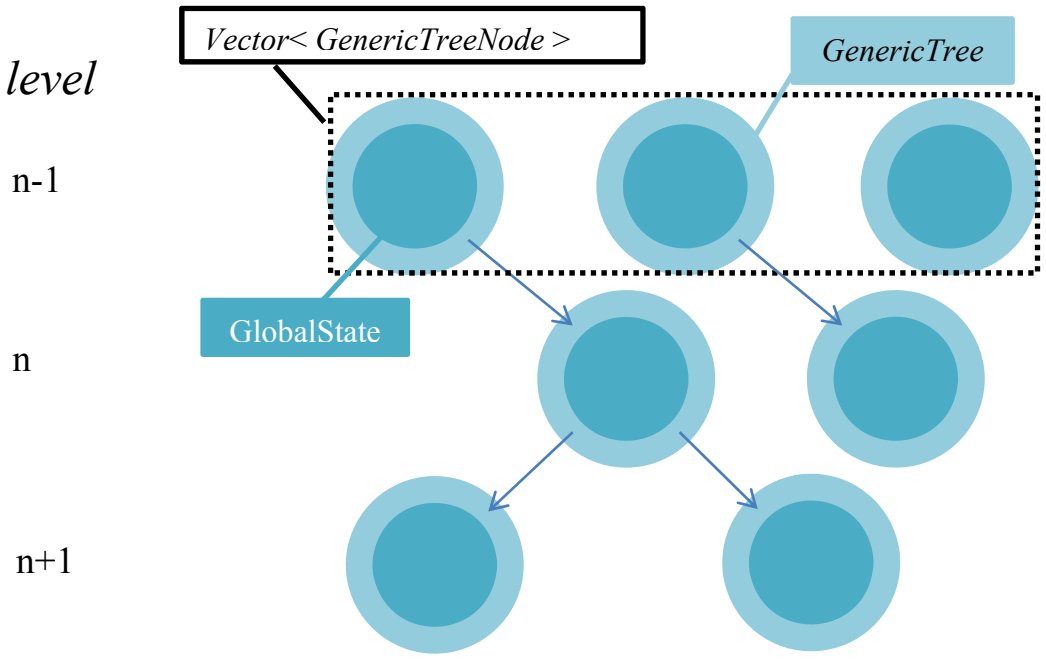


Figure 8-29 *lattice* structure

The diagram demonstrates: The current level is n. The first element of level n has one parent node and two child nodes. The second element of level n has only one parent node.

The procedure to evaluate the predicate in the *LatticeBuilder* class has three steps.

1. Constructing consistent global states
2. Building lattice
3. Predicate evaluation

These steps are separated to different functions.

The *getCGSs* function evaluates the local states and constructs consistent global state with validated local states. It returns a *GlobalState Vector*. All consistent global states are stored in this vector. Figure 8-30 is the work flow of the *getCGSs* function.

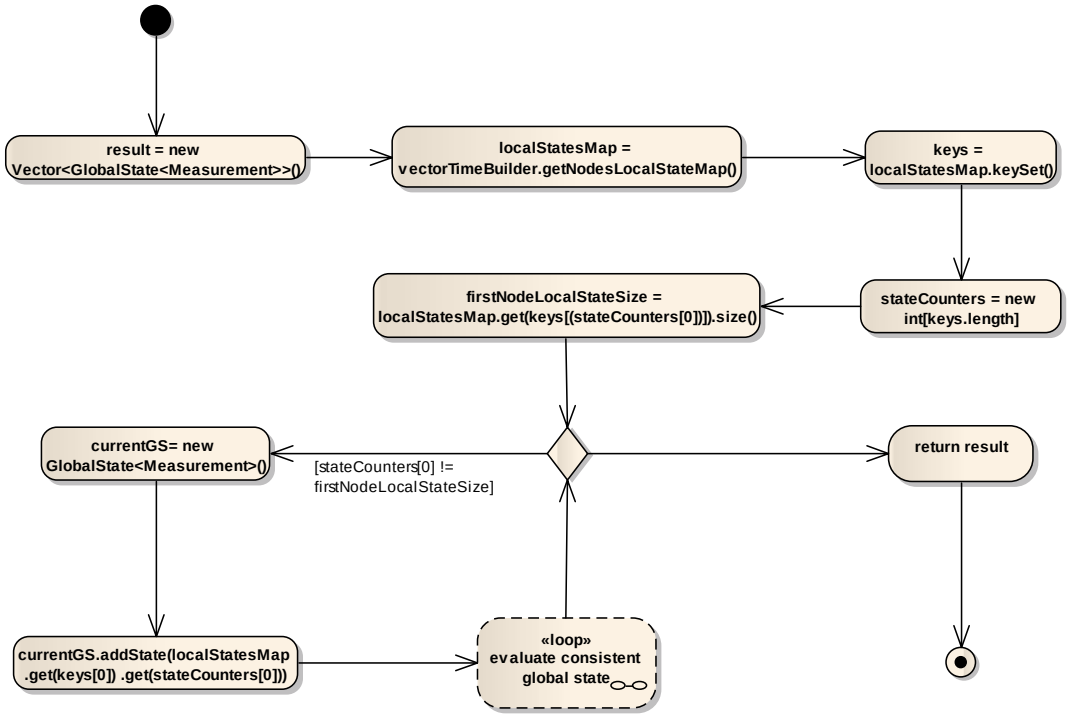


Figure 8-30 the *getCGSs* function activity diagram

Before explaining Figure 8-30, an important algorithm needs to be explained. The *getCGSs* function needs to go through all possible combinations of the local states to evaluate if they are consistent global states. In order to search all local states of all nodes, a "counting" algorithm is used. The "counting" algorithm is similar to counting numbers. It uses an array to store the digits; each element in the array can be considered as a digit. The size of the array is the total number of nodes. Each element of the array is an index counter to go through its corresponding node *LocalState Vector*. A base of a binary system is 2, a base of a octal system is 8 etc. Here the base of each digit (element of the array) is variable. The base of a digit is the total number of local states of the node. If the lower level digit exceeds its base, then the higher level digit increases one. By using such counting system, all possible global states of the combination of local states will be checked.

In Figure 8-30, the variable *result* is the returned *GlobalState Vector*. The HashMap *localStateMap* stores all local states of all nodes. The array *keys* stores all node indexes. The array *stateCounters* is used as a counter as described above ("counting" algorithm) to go through all possible global states constructed by local states. The *Integer firstNodeLocalStateSize* is the size of the *LocalState Vector* mapped by *key[0]* in the *localSTateMap*. It is the base of the highest digit of the counting system. All these variables are initialized or assigned at the start of the work flow. The counting system is implemented by two loops. The first loop is used to control the counting flow. The second loop is used to add the local state of each node to a temporary global state and evaluates this global state.

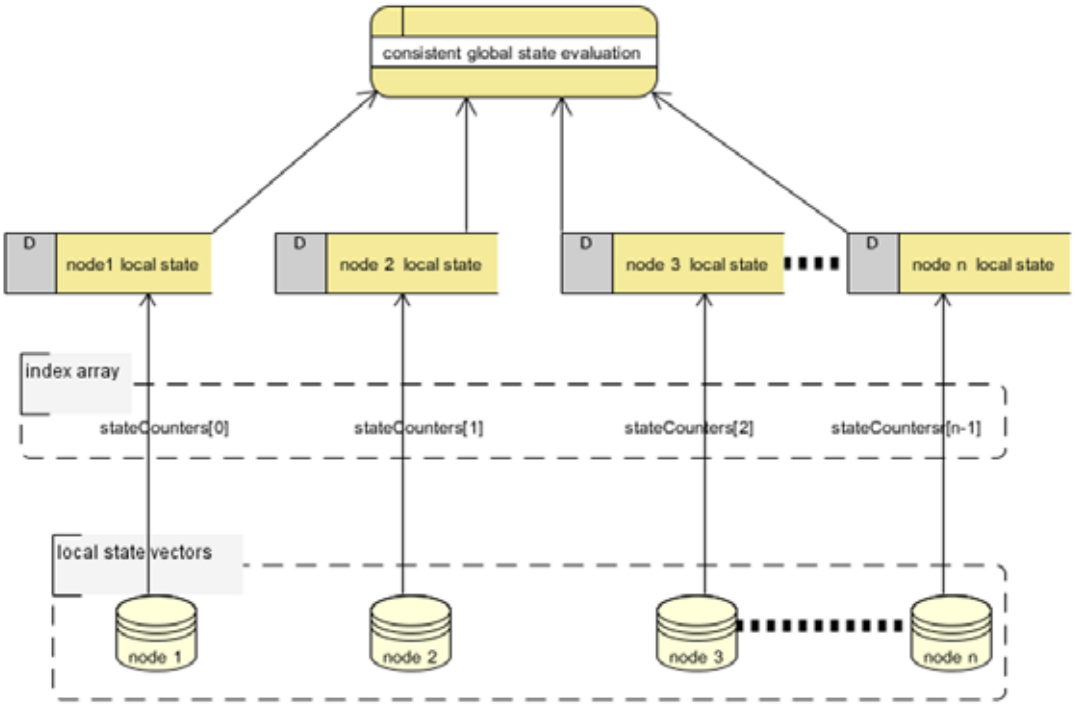


Figure 8-31 counting system structure

The structure of the counting system is illustrated in Figure 8-31. The component *stateCounters[0]* is the highest digit in the counter array. The first loop keeps iterating, when *stateCounters[0]* does not equal to *firstNodeLocalStateSize*.

stateCounters[0] reaching the *firstNodeLocalStateSize* means that the counting is finished. In this loop, a temporary *GlobalState* object *currentGS* is created to hold the local state of each node. After *currentGS* created, the first local state is added to it. The second loop that is an internal loop of the first loop adds the remaining local states to *currentGS*, and evaluates if *currentGS* is consistent. The work flow of this loop is shown in Figure 8-32. It is a for-loop. The counter *i* counts the node index.

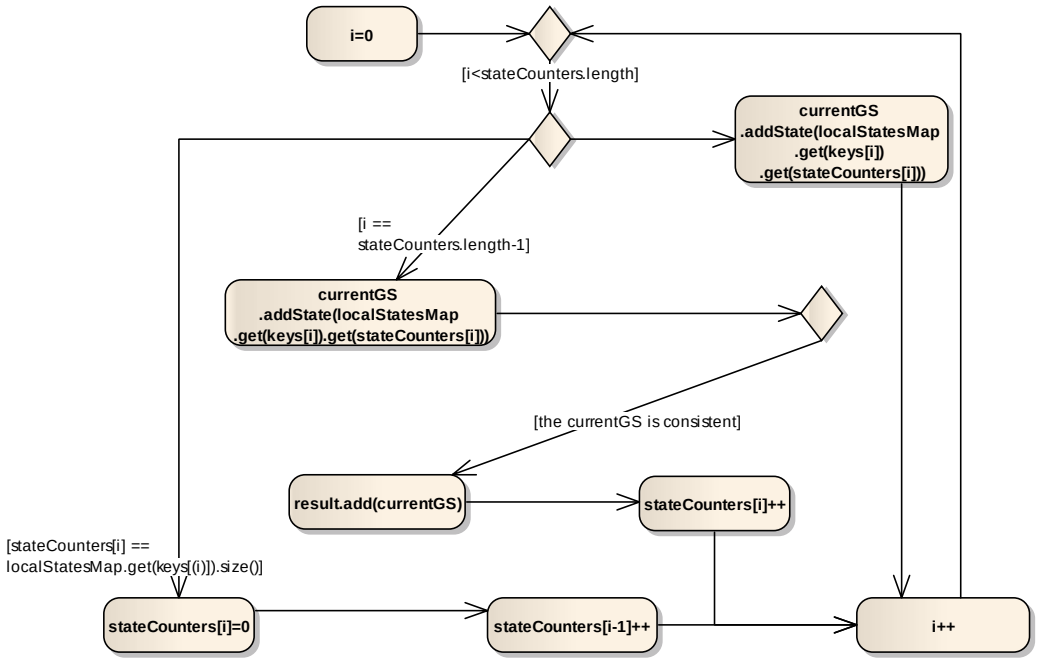


Figure 8-32 work flow of the consistent global state evaluation

The second loop is a for-loop. The counter *i* count the node index. The number of local states in the global state is checked at beginning. If the number of nodes held by *currentGS* is smaller than the total number of nodes (*i<stateCounters.length*), then the loop keeps iterating. There are three conditions in the loop. The first condition (*stateCounters[i] == localStatesMap.get(keys[(i)]).size()*) checks if the digit counter reaches the base, if

it is true, then this digit will be reset to zero (*stateCounters[i]=0*) and the higher level counter increases by one (*stateCounters[i-1]*++). The second condition checks if *currentGS* gets all local states, if it is true, then *currentGS* is checked by its own function *isGSconsistent*, if it consistent, then it is added to *result*. If previous two conditions fail, the third condition adds the local state to *currentGS*.

Depending on the combination equation, the total number of combinations of local state of global state can be as much as

$$l_1 \times l_2 \times l_3 \ldots l_n$$

l is the total number of local states of a node. n is the index number of the node. The combinations depend on the number of nodes and number of local states. Suppose if the node number is 10 and 100 local states are collected for each node, it is 100^{10} possible combinations. The combinations to be evaluated can be a very large number, and also the validated CGSs may need large storage.

After all consistent global states are defined; the execution lattice can be built. The main process to build the execution lattice is shown in Figure 8-33.

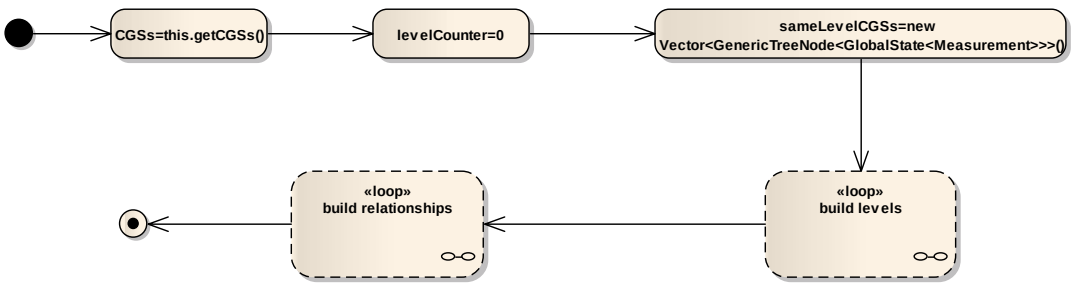


Figure 8-33 main process of building execution lattice

The main process to build the execution lattice is: store all consistent global states to an *GlobalState Vector*, assigning a variable *levelCounter* to count the level, creating a temporary *GenericTreeNode Vector* called *sameLevelCGSs* that is used to store the same level *CGSs*, a loop building the levels of the lattice, and a loop building the relationship between levels.

The work flow of the loop to build the levels is illustrated in Figure 8-34. The loop goes through *CGSs* to add the CGS to corresponding level *GenericTreeNode Vector* of the class variable *lattice*. The exit condition of this loop is when the *CGSs* is empty. In each loop iteration, the level of the CGS is checked (*CGSs.firstElement().getLevel()==levelCounter*) . If the level of the CGS equals to *levelCounter*, then the CGS is removed from *CGSs* and wrapped in a temporary *GenericTreeNode* object *GSnode* that is added to *sameLevelCGSs*. Otherwise a new level and the corresponding *GenericTreeNode Vector* is added to *lattice*; sameLevelCGSs is emptied, so the next level *GenericTreeNode Vector* can be stored.

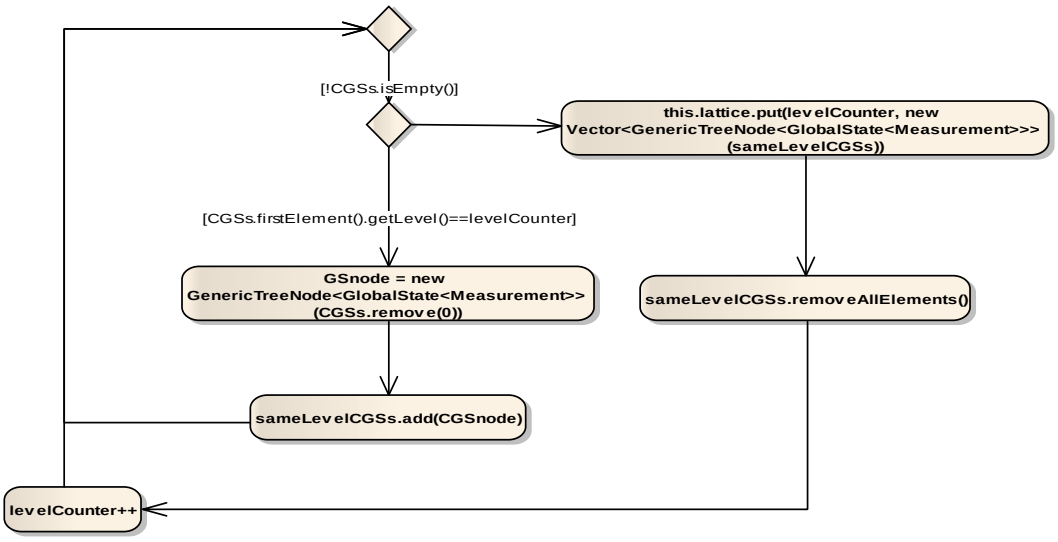


Figure 8-34 work flow build lattice levels

The work flow of the loop to build relationships between parent nodes and child nodes is illustrated in Figure 8-35. The loop goes though each level to relate parent nodes to child nodes. The loop starts with level one (i=1), so level i-1 holds all parent nodes (*parentLevel=i-1*) (*parentSize = this.lattice.get(parentLevel).size()*) and level i holds all child nodes. There is a loop (parent loop) in this loop. The parent loop goes through the parent level nodes to relate the node to its child nodes. There is an inner loop (child loop) in each round of the parent loop. The child loop goes through the child level nodes to relate the node to its parents.

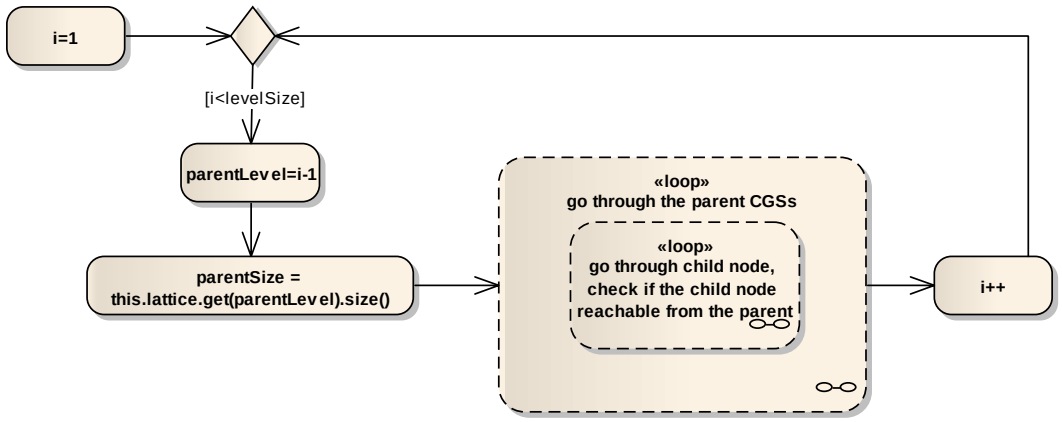


Figure 8-35 work flow to build relationships between parent nodes and child nodes

After the lattice builds up, the *predicate* evaluation can start. There are two evaluation types of the *predicate*. They are *possibly predicate* and *definitely predicate*. The corresponding implementation functions are *possiblyPredicate(Vector<Predicate> inputs)* and *definitelyPredicate(Vector<Predicate> inputs)*. Both of them take *Predicate Vector* (*inputs*) as parameter. A library *Jep* (Nathan Funk 2011) is used for the purpose to compare the state values to the predicate. After the program is compiled, it is hard to evaluate mathematical expressions that are dynamically

defined; particularly if the expression contains the internal variables and external values. *Jep* parses mathematical expression strings and generates the result.

As shown in section 6.4.1 to evaluate *possibly predicate* is easier than evaluating *definitely predicate*. For detecting the *possibly predicate*, it only needs to be proven one global state in the lattice satisfies the predicate. The Predicate function tags all global states that satisfy the predicate. Its execution order is demonstrated in Figure 8-36.

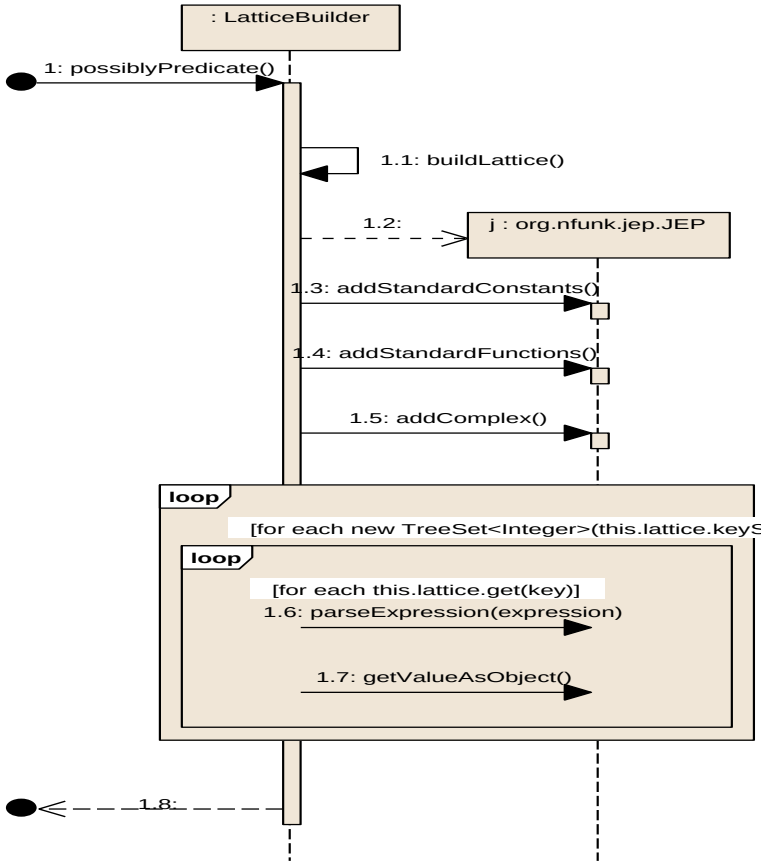


Figure 8-36 the Predicate function sequence diagram

At the beginning of the sequence, the Predicate function calls the *buildLattice()* function to build the lattice. A new object of *JEP j* is created. The steps from 1.3 to 1.5 in Figure 8-36 are initializing *j*. The following two loops go through *lattice*

to evaluate if the global state satisfies the predicate. The external loop goes through the different level. The internal loop goes through the global state vector that is mapped by the level. The step 1.6 shows *j* evaluates the predicate and the step 1.7 is get result from the evaluation. The detail of the work flow in internal loop is demonstrated in Figure 8-37.

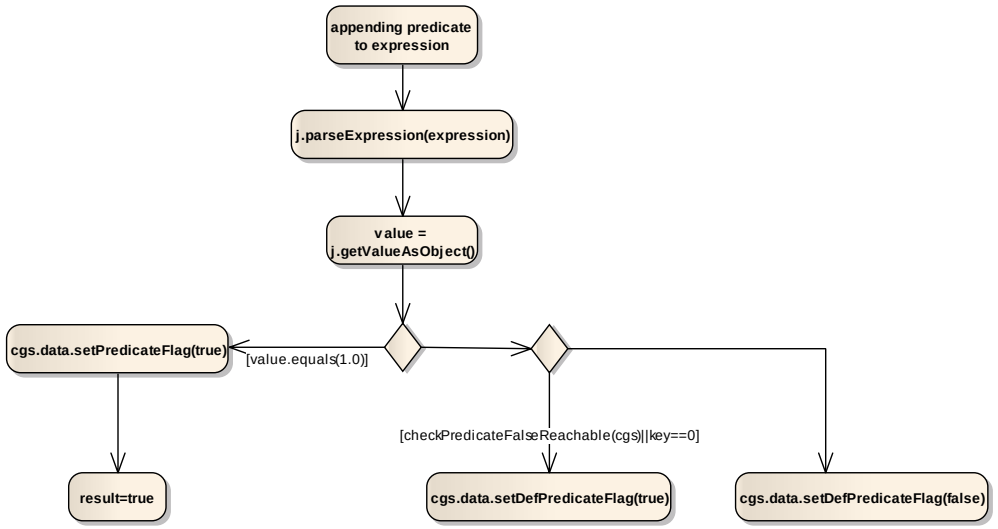


Figure 8-37 work flow of the internal loop of the Predicate function

After the evaluation the result is assigned to *value*. The value of *value* is checked. If the value is 1.0 then it means the global state satisfies the predicate. The return value *result* and the value *predicateFlag* will be set to true. Otherwise it means the global state does not satisfy the predicate. There is a decision here to check if the non-satisfied global state is reachable from the initial state. If it is reachable then the *defPredicateFlag* of the global state is set to true. Otherwise it is set to false. This decision is used for the *definitely predicate* for later. So the *Predicate* function can be used as the sub-function of the *Predicate* function (Figure 8-38).

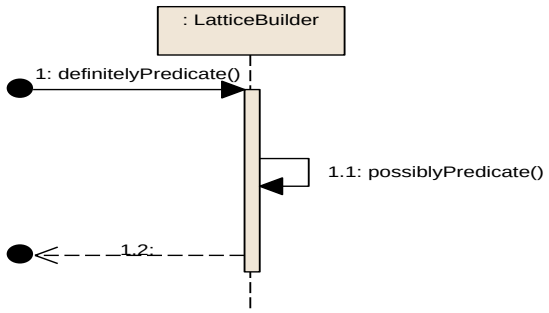


Figure 8-38 sequence diagram for the *Predicate* function

After the *possiblyPredicate* function is invoked, all global states that do not satisfy predicate and is reachable from initial global state are marked (*defPredicateFlag*). As shown in section 6.4.1 the *definitely predicate* detection algorithm, a loop is used to check each level to see if any of them do not contain a global state reachable from initial state without predicate being true.

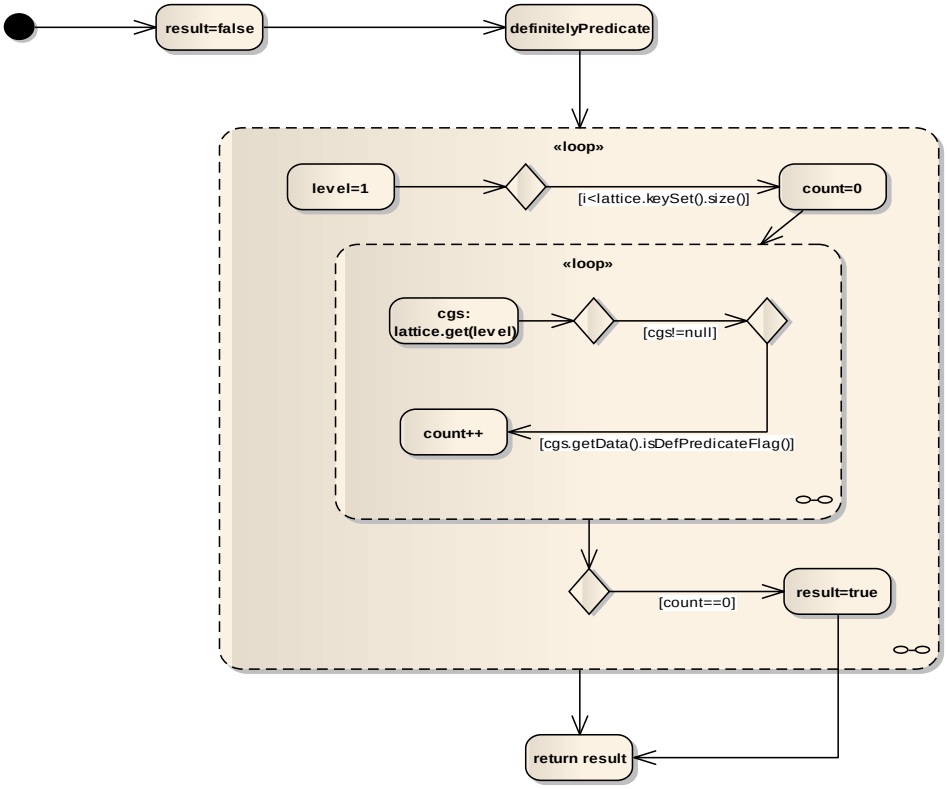


Figure 8-39 the *definitelyPredicate* function work flow

The work flow of the *definitelyPredicate* function is shown in Figure 8-39. The external loop goes through the level. A variable *count* is assigned to zero. It is used to count how many global states marked with *defPredicateFlag*. The internal loop goes through the global states in the level to check if the global state marked with *defPredicateFlag*. If the global state marked with *defPredicateFlag*, then *count* increases one. After the internal loop, *count* is checked. If it is zero, then the predicate is *definitely* true; the function terminates and returns value *result* assigned true. Otherwise it goes to check next level. If all levels checked, and there is no *count* is zero, then the predicate is not *definitely* true. *result* is returned as false.

8.5.4 GraphicGPD

The *GraphicGPD* package gives a GUI to control the predicate evaluation tool and also generates the evaluation result in a graph. It makes the control of the tool easier and the graphic view of the execution lattice makes it more understandable. There are five classes in the *GraphicGPD* package. They are *CGScell, LatticeImageTraslator, GSlatticeFrame, GPDtoolController*, and *InputValueSelector*. Their relationships are demonstrated in Figure 8-40.

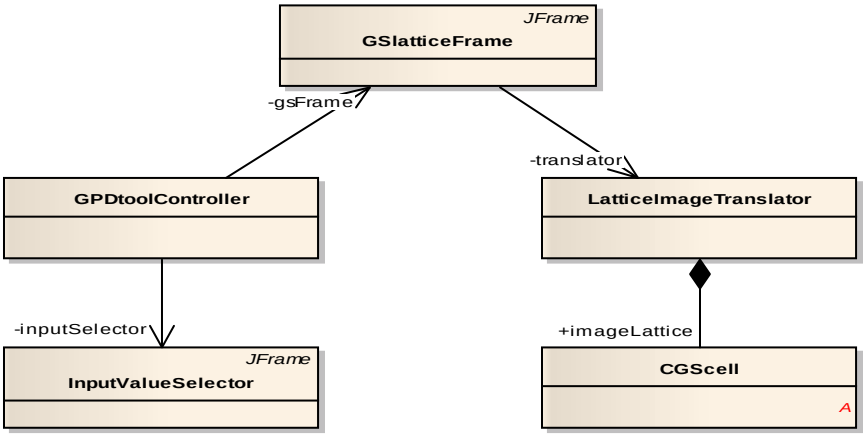


Figure 8-40 *GraphicGPD* package class diagram

The *CGScell* class holds a global state, positioning it on the graphic view of the lattice. It uses different colors to demonstrate if the state satisfies the predicate and the reachable non-satisfied predicate global state from the initial global state. The circles that are shown in Figure 8-41 illustrate the *CGScell* objects.

The *LatticeImageTranslator* class reassembles each level mapped *GlobalState Vector* to *CGScell Vector*. It generates a *BufferedImage* of the lattice.

The *GSlatticeFrame* class draws the image depending on the *BufferedImage* of the lattice. Figure 8-41 shows a lattice frame with a *BufferedImage* embedded.

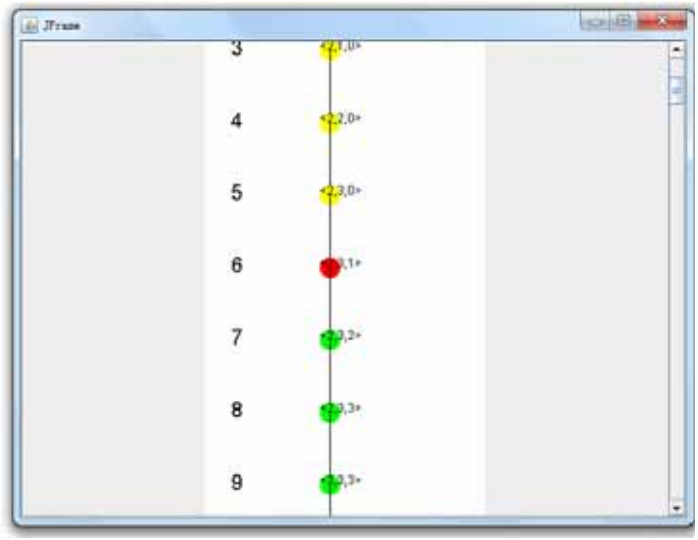


Figure 8-41 Lattice frame

The *GPDtoolController* class gives a GUI to configure the file paths (CANoe log and communication matrix) and select the predicate evaluation type. A *GPDtoolController* dialog is illustrated in Figure 8-42.

The *InputValueSelector* class gives a graphic table to set up the *predicate*s. A *InputValueSelector* dialog is illustrated in Figure 8-43.

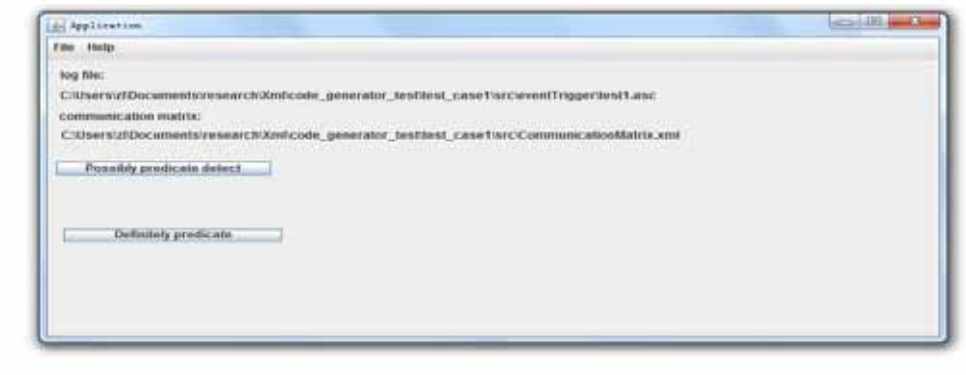


Figure 8-42 *GPDtoolController* dialog

Figure 8-43 *InputValueSelector* dialog

8.6 Conclusion

This chapter described the GPD prototype program design. The prototype includes two main parts: test case generating and global predicate evaluation. For the purpose of rapidly and clearly generating test cases, UML modelling language is used. Depending on the state machine modelled, the state machine can be described in xml format. This xml file is read by a CAPL code generator to parse the XML to CAPL code which simulates the ECU. Running these CAPL codes on CANoe, CANoe logs the CAN messages of the system for the prototype program to evaluate.

The global predicate evaluation program reads the CANoe log file; selects the local state from the file, assigning the vector time; finds out the consistent global states, builds the lattice with these CGSs, and evaluates the predicate. Depending on the total number of node and the local states logged in the CANoe log, the

evaluation process takes a different time. The more node and/or local states, the more time taken. The storage of the CGSs also depends on the communications between the nodes. The more communications, the less CGSs are stored, because more communications make more constraints to evaluate the CGSs.

The result of the evaluation can be graphically viewed. The execution lattice is drawn in a java frame. The predicate situation is demonstrated in different colours.

References

Nathan Funk. Jep Java Math Expression Parser. 2-8-2011.

Vector CANtech, I. Programming With CAPL. 12-14-2004.

Chapter 9 Prototype Testing

9.1 Introduction

This chapter describes test cases to test the prototype global predicate evaluation software. These test cases will verify the functions of the prototype and finally validate the prototype. Each test case includes ECU state diagrams, the values of different states, and the communication matrix. These test cases test simulated ECUs running on the simulated bus and real bus. On the simulated bus, all simulated ECUs run on one computer and they share the single CPU clock; so their clocks are synchronized by the clock of the computer. On the real bus, some simulated ECUs are moved to another computer. They communicate with each other through the real CAN bus. Because these ECUs are running on different computers, the ECUs use different CPU clocks, they are not synchronized by a single clock.

The terms and structures used by the different test cases are very similar, so they will be described first. For better understanding, the first test case will be described with some assistant texts. The terms and structures used by the following test cases will be as same as the first one, so it is not necessary to explain them again.

There are seven test cases in this chapter. Test cases 1 to 4 verifies and validates the prototype. The last three test cases test the performance of the prototype.

In order to make the execution lattice more understandable, some of the results of the test cases are graphically presented.

Table 9-1 gives a general overview of all test cases.

Test case	Number of nodes	Purpose
1	2	Verify and validate prototype
2	3	Verify and validate prototype
3	2	Verify and validate prototype
4	3	Verify and validate prototype
5	4	Test prototype Performance
6	4	Test prototype Performance
7	6	Test prototype Performance

Table 9-1 overview of test cases

The first four test cases are used to verify and validate prototype. Because the verification and validation is done manually, the quantity of nodes is less than three. Otherwise, it is hard to accurately verify and validate the prototype. The first test case verifies the vector clock assignment and building lattice functions. Test case 2 tests high dependence system. Test case 3 tests the effect of environment variables in the state machine. Test case 4 is randomly generated.

The last 3 test cases test the performance of the prototype. Test case 5 tests 4-nodes system performance (randomly chosen). Test case 6 tests non-communication system performance. Test case 7 tests 6-nodes system (randomly chosen).

Due to the time constraints of this research, these test cases cannot fully test the prototype. However, they verify the basic functions, e.g. vector time assignment, consistent global state verification, and building the execution lattice. They also validate the predicate evaluation.

9.2 Term explanation

9.2.1 Terms used on state machine diagram

On the state of the state machine diagram, “set” is for the action *set timer*. Its following variable is the *timer* variable in the CAPL code. The timer variable always starts with letter “t”. “send” is for the action *send message*. The message variable starts with “msg”.

Each “t” or “msg” is followed by letter “n” with number, which means the node with node number. The last number of the variable is the counter of the *timer* variable or the *message* variable. For example the variable “msg_n1_2” means it is a *message* variable and it is the second message sent by node one. There are also another variable starting with “env” which indicates the variable is an environment variable. It is not on the state. It only appears on the transition.

The guard of the transition on the state machine diagram, “on” means timer expired triggers the transition or an environment variable triggers the transition; “rec” means when a message is received the transition is triggered. The variable following “on” or “rec” is the trigger variable.

Figure 9-1 demonstrates the state machine diagram of node 1. There are three state in the node 1 and four transitions. timer *t_n1_1* is set up to 50 milliseconds in *state1*. State 2 sends message *msg_n1_1* with ID hex 100, sets up timer *t_n1_1*, and sets up timer *t_n1_2*. State 3 sends *message msg_n1_2*. The initial state of the state machine is state 1. When the timer *t_n1_1* expires the state machine moves to state 2. When the timer *t_n1_2* expires the state machine moves to state 3. When state3 receives the message *msg_n2_2* the state machine moves to state 2. When the timer *t_n1_1* expires the state machine move to state 1.

9.2.2 Terms used on communication matrix

The first column (*MessageID*) of the communication matrix records the CAN message ID. The second column (*SendNodeNum*) records the index of the sending node. The third column (***receive: nodeNum***) records the index of the receiving node.

Table 9-2 is the communication matrix for the test case 1. There are four messages on the CAN bus. The message with ID 100 is sent by node 1 and received by *node2*. The message with ID 300 is sent by *node2* and received by *node1*. The message with id 101 is sent by *node1* and received by no node. The message with ID 200 is sent by *node2* and received by no node.

9.2.3 Terms on local state table

A local state table shows the different state values of the state machine. The first column (*stateNum*) of the local state table is the index of the state. The other columns (*var1..var10*) are the values of the state variables.

Table 9-3 shows the local state values of each state. In state 1, the value of variable 1 (*var1*) is 11; the value of variable 2 (*var2*) is 11 etc..

9.3 Test case 1

This test case only includes two nodes. Depending on the UML specification, it is easier to manually figure out the time line of the execution. So it is possible to verify if the right vector clock is assigned to the local state. Also it is easier to manually find out the consistent global states and to build the lattice. Using this lattice against the lattice generated by the prototype verifies if the prototype builds the right lattice.

9.3.1 Model explanation

9.3.1.1 State machine diagram

Test case 1 contains two nodes.

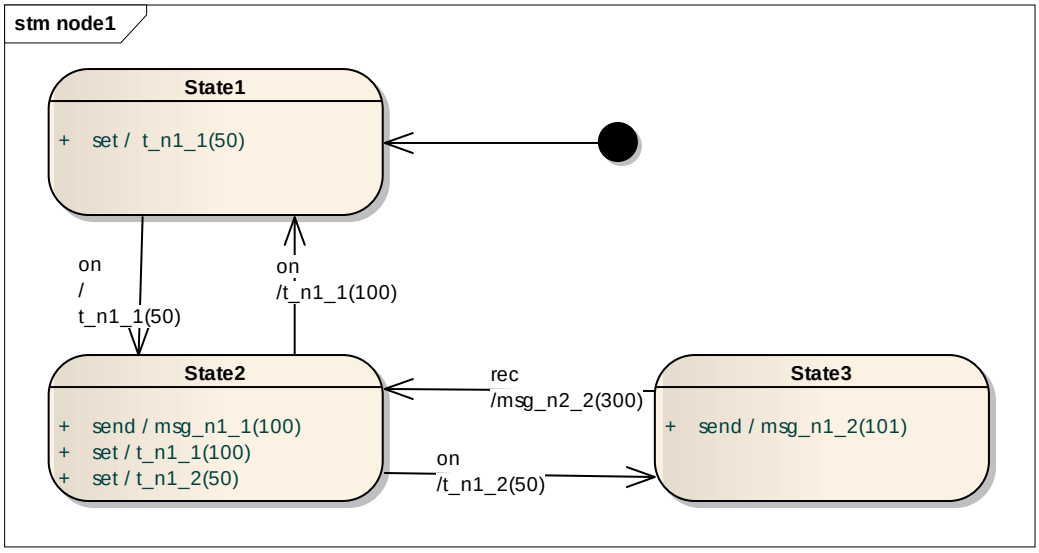


Figure 9-1 test case 1 state machine 1

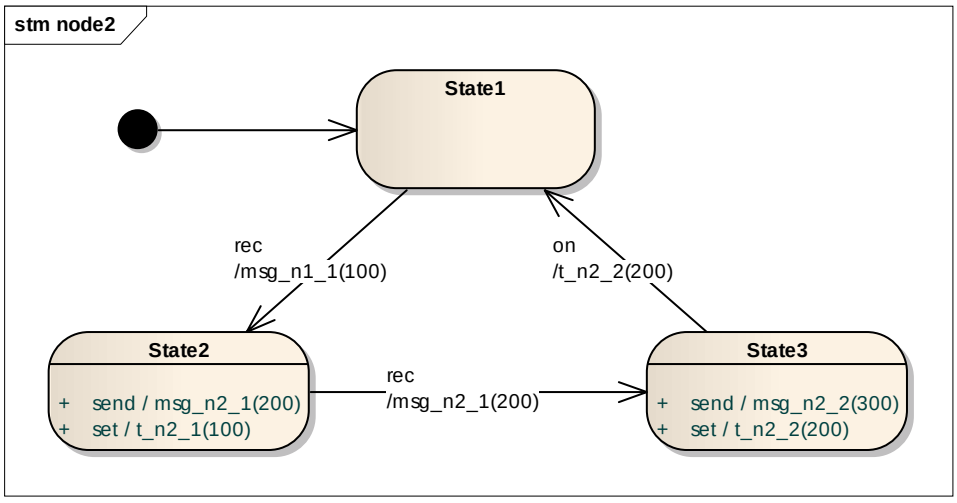


Figure 9-2 test case 1 state machine 2

9.3.1.2 Communication matrix

MessageID	SendNodeNum	receive: nodeNum
100	1	2
300	2	1
101	1	
200	2	

Table 9-2 test case 1 communication matrix

9.3.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	11	11	11	11	0	0	0	0	0
2	12	12	12	12	12	0	0	0	0	0
3	13	13	13	13	13	13	0	0	0	0

Table 9-3 test case 1 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	21	21	21	21	21	0	0	0	0	0
2	22	22	22	22	22	22	0	0	0	0
3	23	23	23	23	23	0	0	0	0	0

Table 9-4 test case 1 *node2* local states

9.3.2 Test different inputs

9.3.2.1 Predicate 1

Predicate expression:

node2.var2 > 21 && *node1*.var2 == 11

var2 of the *node2* is greater than 21 and var1 of node 1 equals to 11

9.3.2.1.1 Simulated system test result

9.3.2.1.1.1 *Local states of each node*

This section shows part of local states of each node.

Vector time	Real time	State	Event
<0,0>	2.36E-04	11 11 11 11 11 0 0 0	internalEvent
<1,0>	0.050246	12 12 12 12 12 0 0 0	internalEvent
<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
<3,0>	0.100234	13 13 13 13 13 13 0 0	internalEvent
<4,0>	0.100368	13 13 13 13 13 13 0 0	sendMsg
<5,5>	0.35105	12 12 12 12 12 0 0 0	receiveMsg
<6,5>	0.351202	12 12 12 12 12 0 0 0	sendMsg
<7,5>	0.45104	11 11 11 11 11 0 0 0	internalEvent
<8,5>	0.55105	12 12 12 12 12 0 0 0	internalEvent
<9,5>	0.551202	12 12 12 12 12 0 0 0	sendMsg
<10,5>	0.601038	13 13 13 13 13 13 0 0	internalEvent

Table 9-5 *node1* local states

Vector time	Real time	State	Event
<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<2,1>	0.050634	22 22 22 22 22 22 0 0	receiveMsg
<2,2>	0.150636	23 23 23 23 23 0 0 0	internalEvent
<2,3>	0.150766	23 23 23 23 23 0 0 0	sendMsg
<2,4>	0.350638	21 21 21 21 21 0 0 0	internalEvent
<2,5>	0.350804	21 21 21 21 21 0 0 0	sendMsg
<6,6>	0.351438	22 22 22 22 22 22 0 0	receiveMsg
<6,7>	0.45144	23 23 23 23 23 0 0 0	internalEvent
<6,8>	0.45157	23 23 23 23 23 0 0 0	sendMsg
<11,9>	0.651442	21 21 21 21 21 0 0 0	receiveMsg
<11,10>	0.651608	21 21 21 21 21 0 0 0	sendMsg
<13,11>	0.652242	22 22 22 22 22 22 0 0	receiveMsg
<13,12>	0.752244	23 23 23 23 23 0 0 0	internalEvent

Table 9-6 *node2* local states

9.3.2.1.1.2 *Consistent global states*

This section shows part of consistent global states found by the prototype. The global states are separated by the blank row.

Global vector time	Local vector time	Real time	State	Event
<0,0>	<0,0>	2.36E-04	11 11 11 11 11 0 0 0	internalEvent
	<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<1,0>	<1,0>	0.050246	12 12 12 12 12 0 0 0	internalEvent
	<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<2,0>	<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
	<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<2,1>	<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
	<2,1>	0.050634	22 22 22 22 22 22 0 0	receiveMsg
<3,0>	<3,0>	0.100234	13 13 13 13 13 13 0 0	internalEvent
	<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<2,2>	<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
	<2,2>	0.150636	23 23 23 23 23 0 0 0	internalEvent
<3,1>	<3,0>	0.100234	13 13 13 13 13 13 0 0	internalEvent
	<2,1>	0.050634	22 22 22 22 22 22 0 0	receiveMsg
<4,0>	<4,0>	0.100368	13 13 13 13 13 13 0 0	sendMsg
	<0,0>	4.76E-04	21 21 21 21 21 0 0 0	internalEvent
<2,3>	<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
	<2,3>	0.150766	23 23 23 23 23 0 0 0	sendMsg
<3,2>	<3,0>	0.100234	13 13 13 13 13 13 0 0	internalEvent
	<2,2>	0.150636	23 23 23 23 23 0 0 0	internalEvent
<4,1>	<4,0>	0.100368	13 13 13 13 13 13 0 0	sendMsg
	<2,1>	0.050634	22 22 22 22 22 22 0 0	receiveMsg
<2,4>	<2,0>	0.050398	12 12 12 12 12 0 0 0	sendMsg
	<2,4>	0.350638	21 21 21 21 21 0 0 0	internalEvent
<3,3>	<3,0>	0.100234	13 13 13 13 13 13 0 0	internalEvent
	<2,3>	0.150766	23 23 23 23 23 0 0 0	sendMsg

Table 9-7 global state

9.3.2.1.1.3 *General result (possibly predicate)*

Total local states of *node1*: 61
Total local states of *node2*: 44
Total number global states: 234
Possibly predicate: true
Running time cost: 346ms

Figure 9-5 shows the graphic result of the test case 1 predicate 1 simulated bus possibly predicate detection. The red and the green circles are the global states of the execution lattice. The red circle means the global state satisfies the predicate. The green circle means the global state does not satisfy the predicate. The numbers on the circle is the vector time of the global state. The numbers on the right of the lattice are the level of the lattice. Figure 9-3 shows the global state with the global vector time <2,3>. The dialog above the lattice is the detail about the global state; it tells the predicate flag is false, so the colour of this global state is green. Figure 9-4 shows the example of the colour of the global state that satisfies the predicate is red (predicate flag is true).

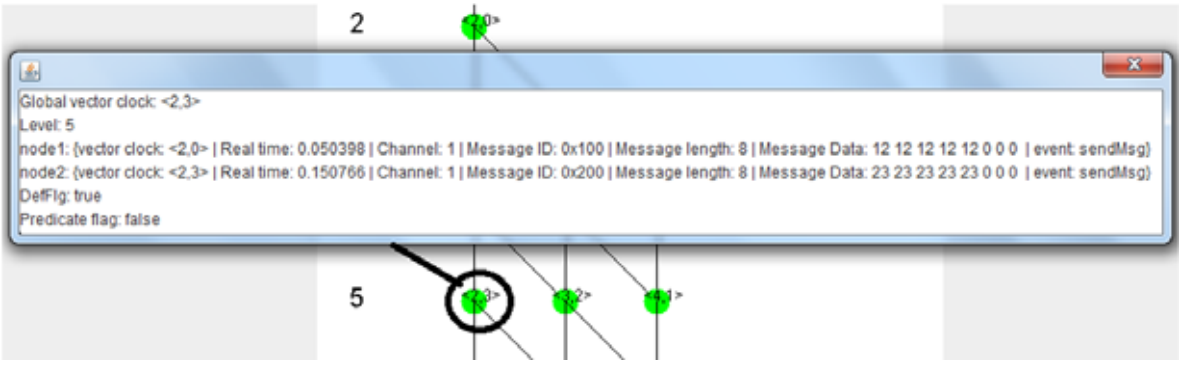


Figure 9-3 the global state of the lattice does not satisfy the predicate.

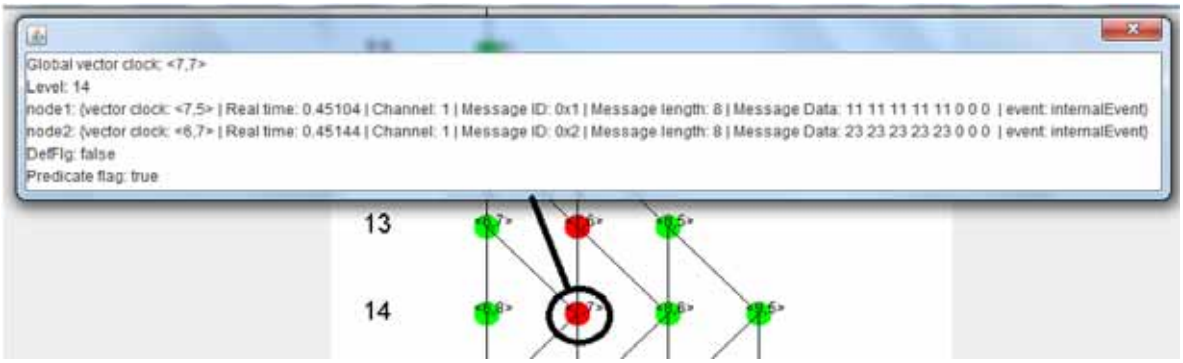


Figure 9-4 the global state of the lattice satisfies the predicate.

For the graphic result of the definitely predicate detection, the global state which is reachable from initial state without predicate being true is coloured yellow as shown in Figure 9-6.

To use graph can clearly show the predicate detection algorithm and result of the predicate evaluation. The more consistent global state the bigger of the graph.

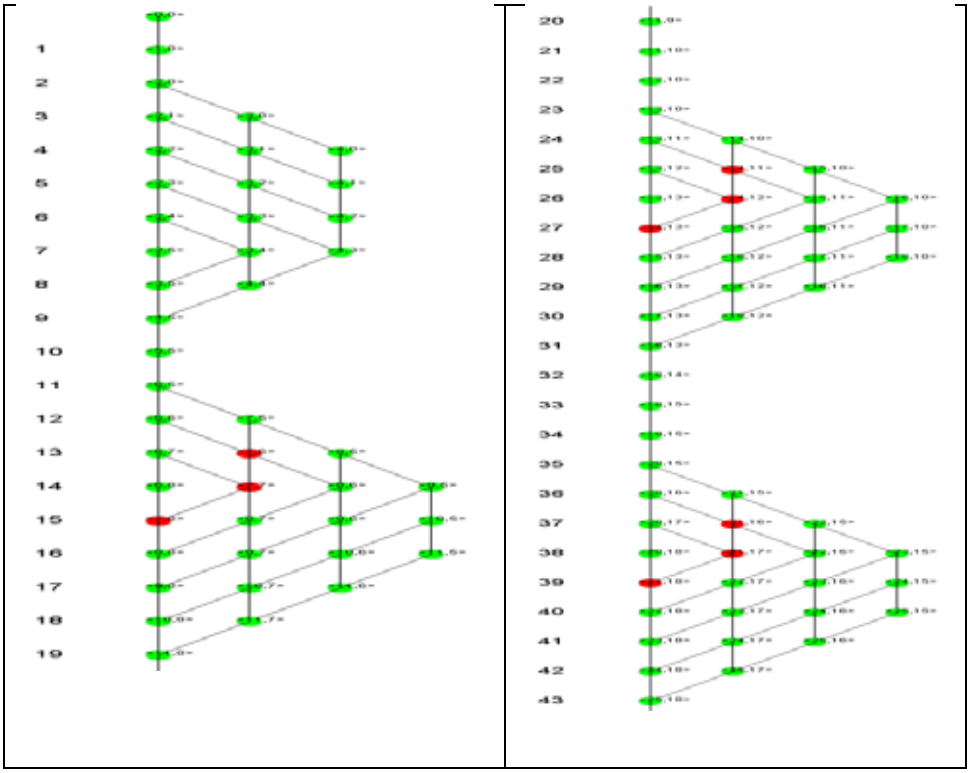


Figure 9-5 test case 1 predicate 1 simulated bus possibly predicate detection graphic result

9.3.2.1.1.4 *General result (Definitely predicate)*

Total local states of *node1*: 61
Total local states of *node2*: 44
Total number global states: 234
Definitely predicate: false
Running time cost: 311ms

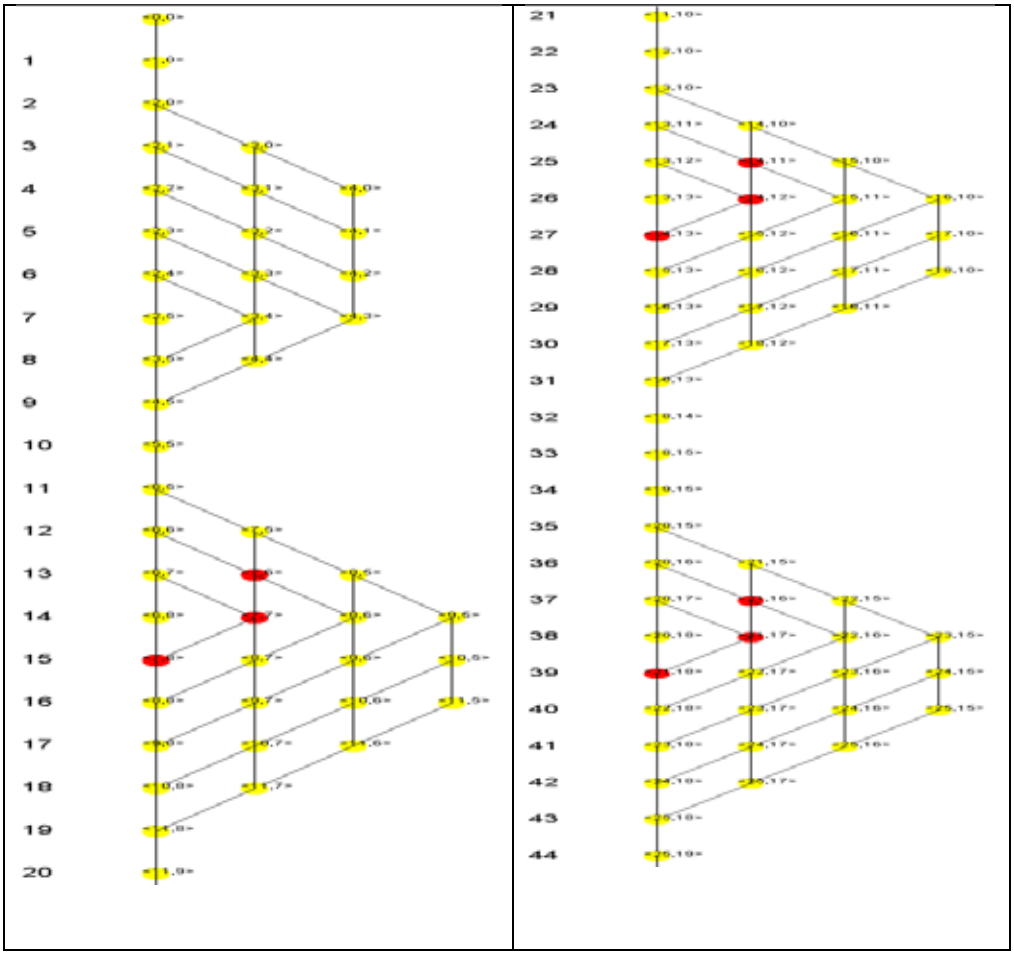


Figure 9-6 test case 1 predicate 1 simulated bus possibly definitely detection graphic result

9.3.2.1.2 Real system test result

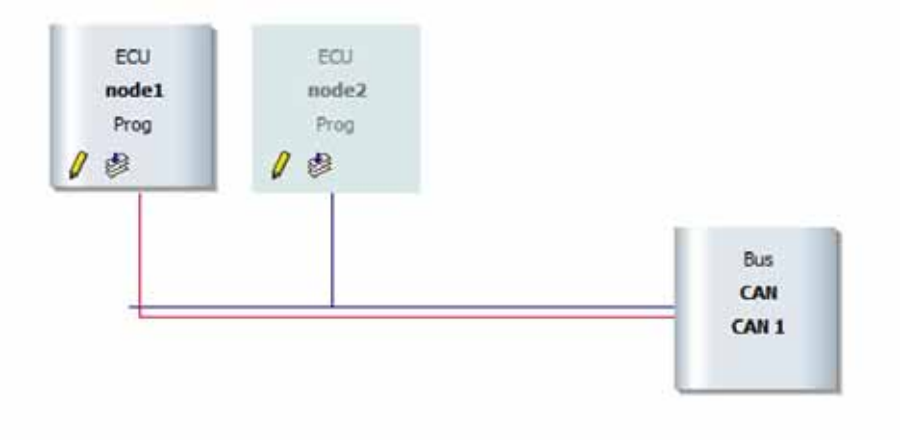


Figure 9-7 test case1 predicate 1 real bus configuration

For the real bus running the test case, the configuration of CANoe is show in Figure 9-7. In the diagram, *node2* is paler than *node1*. This means *node2* simulated by CANoe is deactivated from the CAN bus. *node2* should be replaced by the same function ECU on the real CAN bus.

node1 runs on laptop with CANoe . *node2* is moved to another laptop which runs CANalyzer. CANalyzer is the universal analysis tool for networks and distributed systems. It is pretty similar to CANoe. The CAPL code (*node2*) can be run by CANalyzer as well as CANoe. In order to send CAN messages on the real bus, CANalyzer has to be used with a PC card CANcardXL. This card is connected to the real bus and sends the CAN message generated by *node2* to the real bus.

9.3.2.1.2.1 *Local state of each node*

Vector time	Real time	State	Event
<0,0>	0.075945	11 11 11 11 11 0 0 0	internalEvent
<1,0>	0.076207	12 12 12 12 12 0 0 0	internalEvent
<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
<3,0>	0.100968	13 13 13 13 13 13 0 0	internalEvent
<4,0>	0.101118	13 13 13 13 13 13 0 0	sendMsg
<5,5>	0.37795	12 12 12 12 12 0 0 0	receiveMsg
<6,5>	0.378118	12 12 12 12 12 0 0 0	sendMsg
<7,5>	0.477358	11 11 11 11 11 0 0 0	internalEvent
<8,5>	0.577498	12 12 12 12 12 0 0 0	internalEvent
<9,5>	0.577666	12 12 12 12 12 0 0 0	sendMsg
<10,5>	0.62741	13 13 13 13 13 13 0 0	internalEvent

Table 9-8 *node1* local states

<0,0>	0.075707	21 21 21 21 21 0 0 0	internalEvent
<2,1>	0.076778	22 22 22 22 22 22 0 0	receiveMsg
<2,2>	0.176898	23 23 23 23 23 0 0 0	internalEvent
<2,3>	0.177051	23 23 23 23 23 0 0 0	sendMsg
<2,4>	0.376839	21 21 21 21 21 0 0 0	internalEvent
<2,5>	0.377027	21 21 21 21 21 0 0 0	sendMsg
<6,6>	0.378454	22 22 22 22 22 22 0 0	receiveMsg
<6,7>	0.478898	23 23 23 23 23 0 0 0	internalEvent
<6,8>	0.47905	23 23 23 23 23 0 0 0	sendMsg
<11,9>	0.678916	21 21 21 21 21 0 0 0	receiveMsg
<11,10>	0.679104	21 21 21 21 21 0 0 0	sendMsg
<13,11>	0.680489	22 22 22 22 22 22 0 0	receiveMsg
<13,12>	0.780937	23 23 23 23 23 0 0 0	internalEvent

Table 9-9 *node2* local states

9.3.2.1.2.2 *Consistent global states*

Global vector time	Local vector time	Real time	State	Event
<0,0>	<0,0>	7.59E-02	11 11 11 11 11 0 0 0	internalEvent
	<0,0>	7.57E-02	21 21 21 21 21 0 0 0	internalEvent
<1,0>	<1,0>	0.076207	12 12 12 12 12 0 0 0	internalEvent
	<0,0>	7.57E-02	21 21 21 21 21 0 0 0	internalEvent
<2,0>	<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
	<0,0>	7.57E-02	21 21 21 21 21 0 0 0	internalEvent
<2,1>	<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
	<2,1>	0.076778	22 22 22 22 22 22 0 0	receiveMsg
<3,0>	<3,0>	0.100968	13 13 13 13 13 13 0 0	internalEvent
	<0,0>	7.57E-02	21 21 21 21 21 0 0 0	internalEvent
<2,2>	<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
	<2,2>	0.176898	23 23 23 23 23 0 0 0	internalEvent
<3,1>	<3,0>	0.100968	13 13 13 13 13 13 0 0	internalEvent
	<2,1>	0.076778	22 22 22 22 22 22 0 0	receiveMsg
<4,0>	<4,0>	0.101118	13 13 13 13 13 13 0 0	sendMsg
	<0,0>	7.57E-02	21 21 21 21 21 0 0 0	internalEvent
<2,3>	<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
	<2,3>	0.177051	23 23 23 23 23 0 0 0	sendMsg
<3,2>	<3,0>	0.100968	13 13 13 13 13 13 0 0	internalEvent
	<2,2>	0.176898	23 23 23 23 23 0 0 0	internalEvent
<4,1>	<4,0>	0.101118	13 13 13 13 13 13 0 0	sendMsg
	<2,1>	0.076778	22 22 22 22 22 22 0 0	receiveMsg
<2,4>	<2,0>	0.076375	12 12 12 12 12 0 0 0	sendMsg
	<2,4>	0.376839	21 21 21 21 21 0 0 0	internalEvent
<3,3>	<3,0>	0.100968	13 13 13 13 13 13 0 0	internalEvent
	<2,3>	0.177051	23 23 23 23 23 0 0 0	sendMsg
<4,2>	<4,0>	0.101118	13 13 13 13 13 13 0 0	sendMsg
	<2,2>	0.176898	23 23 23 23 23 0 0 0	internalEvent

Table 9-10 global states

9.3.2.1.2.3 *General result (possibly predicate)*

Total local states of *node1*: 141
Total local states of *node2*: 104
Total number global states: 542
Possibly predicate: true
Running time cost: 356ms

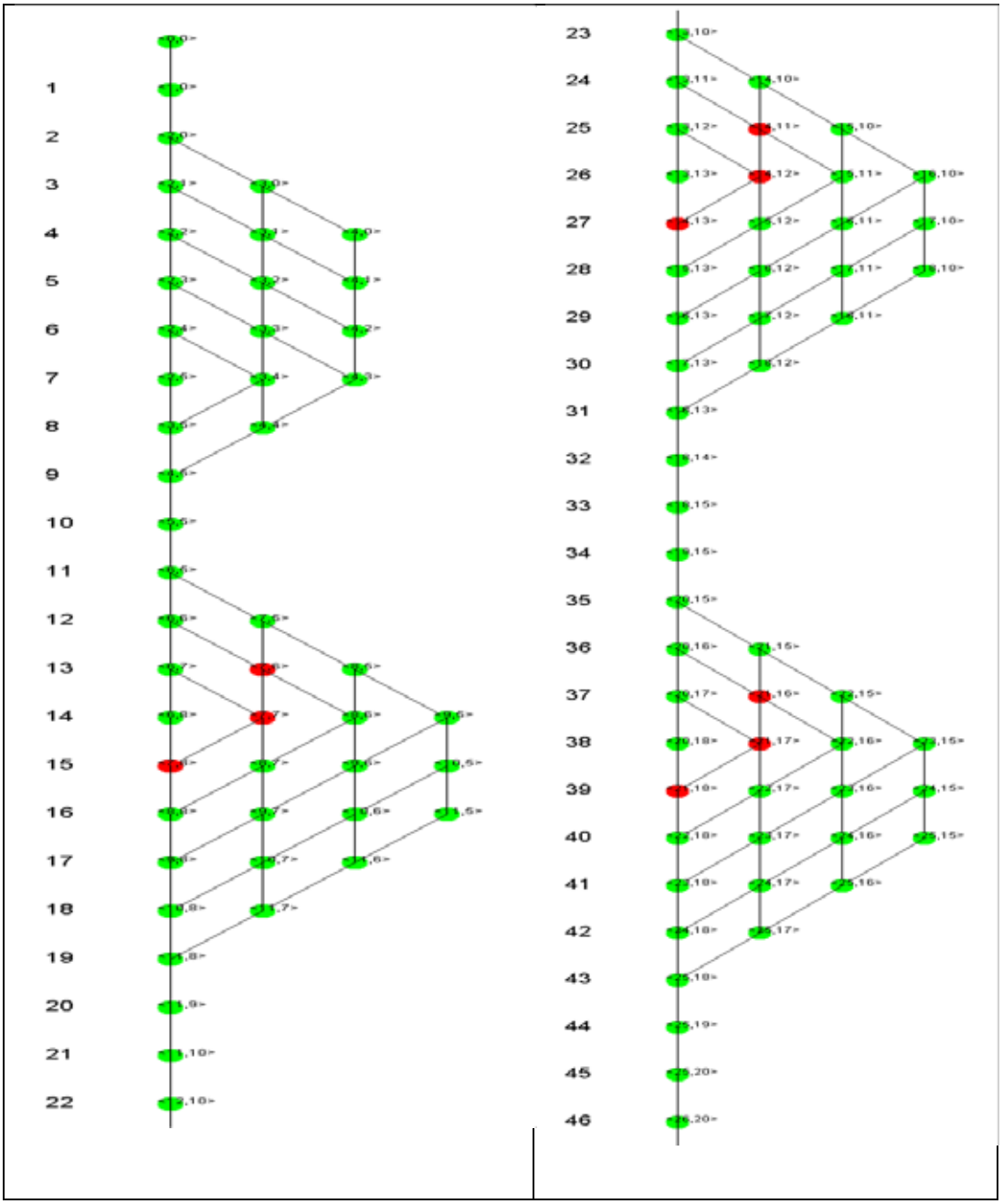


Figure 9-8 test case 1 predicate 1 real bus possibly predicate detection graphic result

9.3.2.1.2.4 *General result (Definitely predicate)*

Total local states of *node1*: 420
Total local states of *node2*: 302
Total number global states: 1626
Definitely predicate: false
Running time cost: 637ms

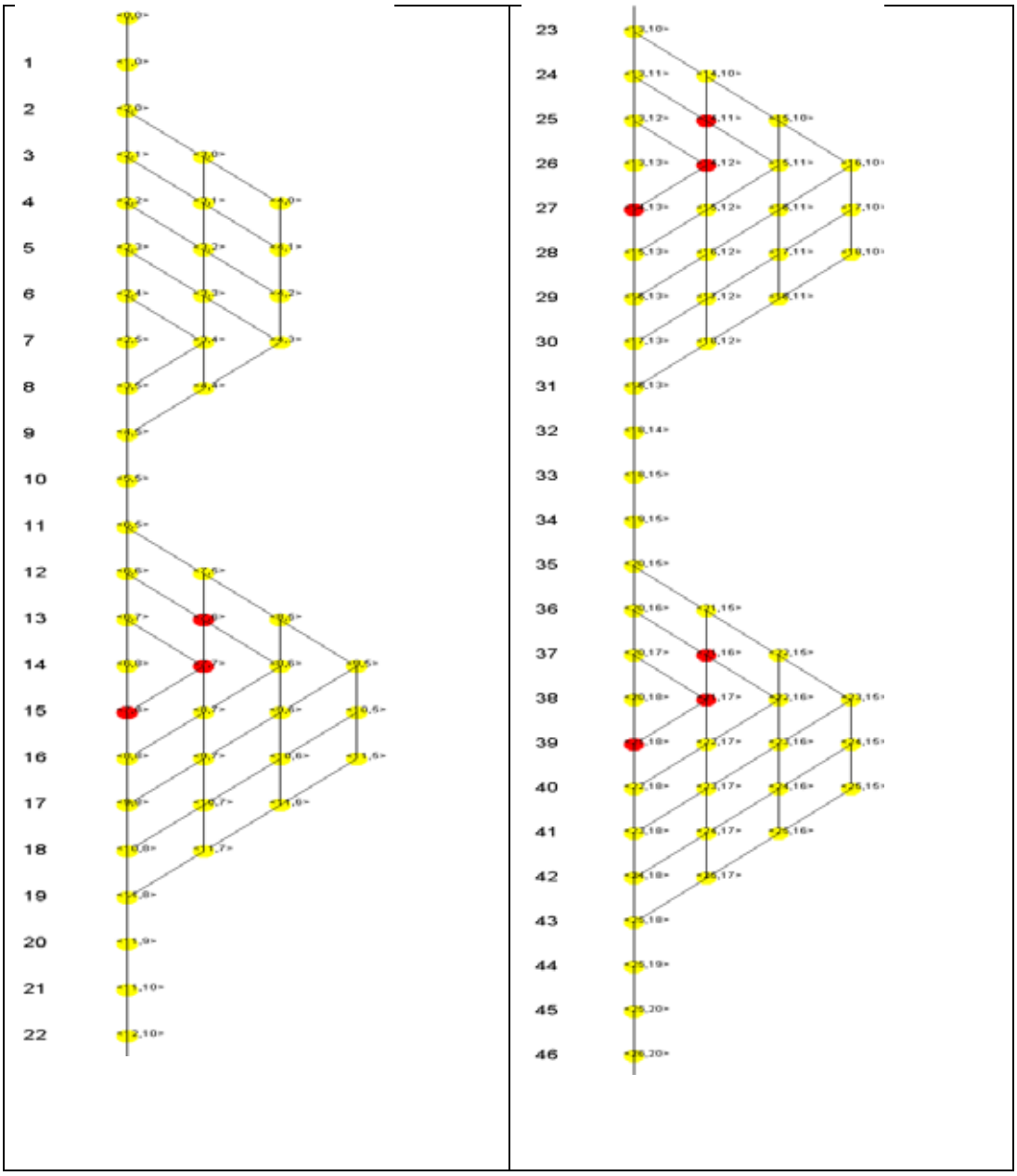


Figure 9-9 test case 1 predicate 1 real bus definitely predicate detection graphic result

9.3.3 Result analysis

The vector clock is the fundamental element for the GPD. So at the beginning, the verification of the function to assign the vector clock is essential.

For assigning the vector clock, if the event does not affect the node state change, the vector clock won't increase. The only exception is the event of sending message. Based on this rule, the vector clock is assigned.

As shown in the state diagram in section 9.3.1.1, the execution of the test case is manually generated as shown in Figure 9-10. In the diagram, the arrow is the message passing on the CAN bus. Dashed blue arrow message is a message that does not affect any change of any node. E.g. *msg_n1_ 2* is not received by any node, so it won't affect any node's state changing. *msg_n1_1* is received by *node2*. But if *node2* is not in *State1*, *node2* won't change its state.

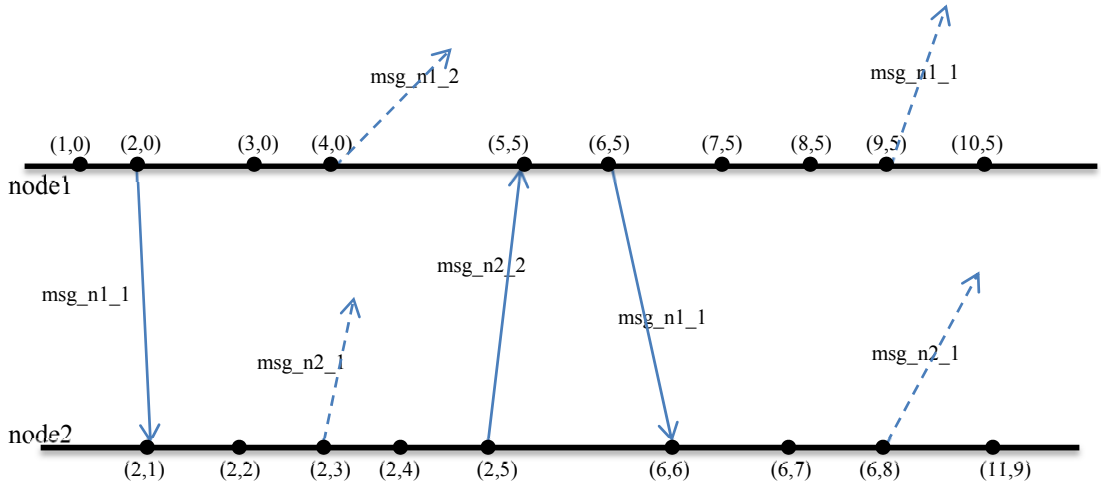


Figure 9-10 test case 1 execution

As shown in the result in section 9.3.2.1.1.1 , the order of vector times of *node1*is <0,0>, <1,0>, <2,0>, <3,0>, <4,0>, <5,5>, <6,5>, <7,5>, <8,5>, <9,5>, <10,5>. This order matches the order of the execution diagram, and the events that these

vector times relate to match the events on the execution diagram. For *node2*, all its vector time order and events match the execution diagram. So for this test case, the function of assigning vector clocks is verified.

The next step is to verify if the global state in the lattice is consistent. Depending on section 7.2.2 and the data in section 9.3.2.1.1.2 that shows the consistent global states that are evaluated by the prototype, the consistent global state evaluation function can be verified. The following example is how to manually verify the function of evaluating global state. The global states with vector time <4,1> on Table 9-7 includes two local states. The vector times of these local states are <4,0> and <2,1>. <4,0> is the vector time of *node1*. <2,1> is the vector time of *node2*. The first element of the vector time records the time of *node1*. The second element of the vector time records the time of *node2*. Manually doing the comparison 4>2 and 0<1, the result is true, so the global state is consistent. The other global states are manually checked by the same method. For this test case, the result of verifying the function of evaluating the consistent global state is success.

The lattice can be visually checked by the lattice diagram (e.g. Figure 9-5, Figure 9-7). For two reachable global states, only one component of the vector time of the parent node is one smaller than the corresponding component of the vector time of the child node. All global states in the lattice diagram match this rule.

The last step is to validate prototype software. To validate the prototype involves going through the global state table, to manually check if the global state satisfies the predicate and to manually mark each check result to the global state. Using this checking result against the result generated by the prototype, if two results match each other, then the prototype is validated; otherwise the prototype is not validated.

In this test case, the expression of predicate 1 is *node2*.var2 > 21 && *node1*.var2 == 11. Going through Table 9-7, in the first global state of the table, *node2*.var2 is 21. It does not satisfy the first expression. *node1*.var2 satisfies the second expression. Because the predicate is a conjunction of the two expressions, the global state does not satisfy the predicate. This global state should be marked as false. The following global states are checked in the same way. After the whole table is checked, each global state is marked with a true or false flag. Using this checked table against Figure 9-5 and Figure 9-6 validates the prototype. The result of the validation is successful for this test case.

To verify the result of the real system, the same result is achieved as for the simulated system. For this test case, it is proved that the prototype can be applied on both systems (simulated and real).

9.4 Test case 2

This test case consists of 3 state machine nodes. One state machine depends on another's message. Each state machine is tightly related to each other.

9.4.1 Model explanation

9.4.1.1 State machine diagram

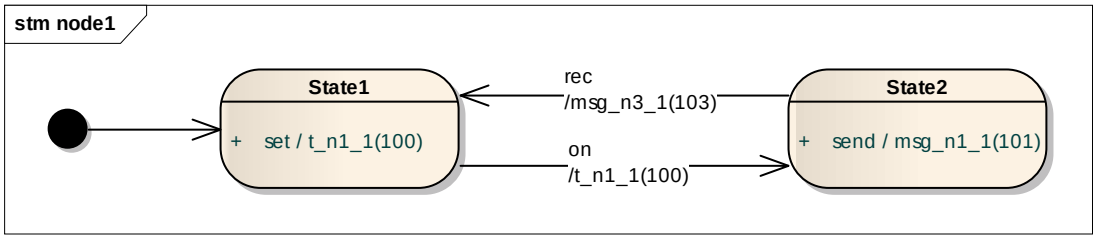


Figure 9-11 test case 2 state machine 1

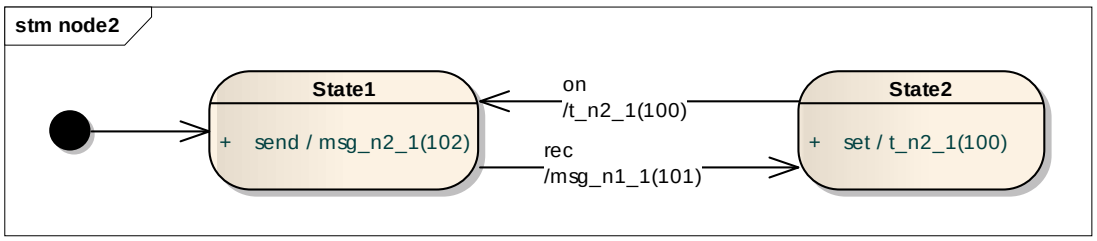


Figure 9-12 test case 2 state machine 2

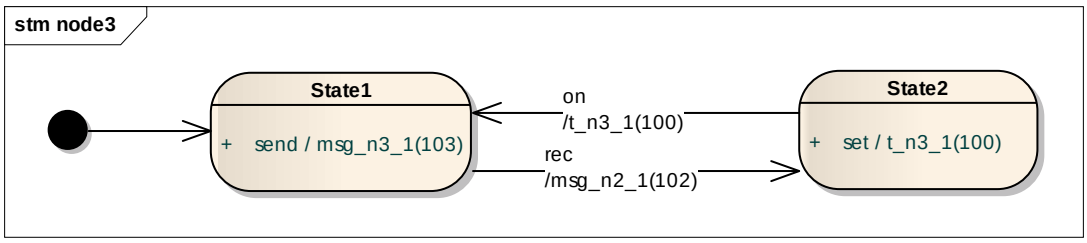


Figure 9-13 test case 2 state machine 3

9.4.1.2 Communication matrix

MessageID	SendNodeNum	receive: nodeNum
101	1	2
102	2	3
103	3	1

Table 9-11 test case 2 communication matrix

9.4.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	11	11	11	11	0	0	0	0	0
2	12	12	12	0	0	0	0	0	0	0

Table 9-12 test case 2 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	21	21	21	21	0	0	0	0	0	0
2	22	22	22	0	0	0	0	0	0	0

Table 9-13 test case 2 *node2* local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	31	31	31	0	0	0	0	0	0	0
2	32	32	32	32	0	0	0	0	0	0

Table 9-14 test case 2 node 3 local states

9.4.2 Test different inputs

9.4.2.1 Predicate 1

node2.var2==11|| node3.var2==33

9.4.2.1.1 Simulated system test result

9.4.2.1.1.1 *Local states of each node*

Vector time	Real time	State	Event
<0,0,0>	2.36E-04	11 11 11 11 11 0 0 0	internalEvent
<1,0,0>	0.100248	12 12 12 0 0 0 0 0	internalEvent
<2,0,0>	0.100382	12 12 12 0 0 0 0 0	sendMsg
<3,3,3>	0.30138	11 11 11 11 11 0 0 0	receiveMsg
<4,3,3>	0.401392	12 12 12 0 0 0 0 0	internalEvent
<5,3,3>	0.401526	12 12 12 0 0 0 0 0	sendMsg
<6,6,6>	0.602524	11 11 11 11 11 0 0 0	receiveMsg
<7,6,6>	0.702536	12 12 12 0 0 0 0 0	internalEvent
<8,6,6>	0.70267	12 12 12 0 0 0 0 0	sendMsg
<9,9,9>	0.903668	11 11 11 11 11 0 0 0	receiveMsg
<10,9,9>	1.00368	12 12 12 0 0 0 0 0	internalEvent

Table 9-15 node1 local states

Vector time	Real time	State	Event
<0,0,0>	4.78E-04	21 21 21 21 0 0 0 0	internalEvent
<2,1,0>	0.100626	22 22 22 0 0 0 0 0	receiveMsg
<2,2,0>	0.200624	21 21 21 21 0 0 0 0	internalEvent
<2,3,0>	0.20076	21 21 21 21 0 0 0 0	sendMsg
<5,4,3>	0.40177	22 22 22 0 0 0 0 0	receiveMsg
<5,5,3>	0.501768	21 21 21 21 0 0 0 0	internalEvent
<5,6,3>	0.501904	21 21 21 21 0 0 0 0	sendMsg
<8,7,6>	0.702914	22 22 22 0 0 0 0 0	receiveMsg
<8,8,6>	0.802912	21 21 21 21 0 0 0 0	internalEvent
<8,9,6>	0.803048	21 21 21 21 0 0 0 0	sendMsg
<11,10,9>	1.004058	22 22 22 0 0 0 0 0	receiveMsg
<11,11,9>	1.104056	21 21 21 21 0 0 0 0	internalEvent

Table 9-16 node2 local states

Vector time	Real time	State	Event
<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<2,3,1>	0.201008	32 32 32 32 0 0 0 0	receiveMsg
<2,3,2>	0.30101	31 31 31 0 0 0 0 0	internalEvent
<2,3,3>	0.301144	31 31 31 0 0 0 0 0	sendMsg
<5,6,4>	0.502152	32 32 32 32 0 0 0 0	receiveMsg
<5,6,5>	0.602154	31 31 31 0 0 0 0 0	internalEvent
<5,6,6>	0.602288	31 31 31 0 0 0 0 0	sendMsg
<8,9,7>	0.803296	32 32 32 32 0 0 0 0	receiveMsg
<8,9,8>	0.903298	31 31 31 0 0 0 0 0	internalEvent
<8,9,9>	0.903432	31 31 31 0 0 0 0 0	sendMsg
<11,12,10>	1.10444	32 32 32 32 0 0 0 0	receiveMsg
<11,12,11>	1.204442	31 31 31 0 0 0 0 0	internalEvent

Table 9-17 node3 local states

9.4.2.1.1.2 *Consistent global states*

Global vector time	Local vector time	Real time	State	Event
<0,0,0>	<0,0,0>	2.36E-04	11 11 11 11 11 0 0 0	internalEvent
	<0,0,0>	4.78E-04	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<1,0,0>	<1,0,0>	1.00E-01	12 12 12 0 0 0 0 0	internalEvent
	<0,0,0>	4.78E-04	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<2,0,0>	<2,0,0>	0.100382	12 12 12 0 0 0 0 0	sendMsg
	<0,0,0>	4.78E-04	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<2,1,0>	<2,0,0>	0.100382	12 12 12 0 0 0 0 0	sendMsg
	<2,1,0>	1.01E-01	22 22 22 0 0 0 0 0	receiveMsg
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<2,2,0>	<2,0,0>	0.100382	12 12 12 0 0 0 0 0	sendMsg
	<2,2,0>	0.200624	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent
<2,3,0>	<2,0,0>	0.100382	12 12 12 0 0 0 0 0	sendMsg
	<2,3,0>	0.20076	21 21 21 21 0 0 0 0	sendMsg
	<0,0,0>	7.28E-04	31 31 31 0 0 0 0 0	internalEvent

Table 9-18 global states

9.4.2.1.1.3 *General result (possibly predicate)*

Total local states of *node1*: 36
Total local states of *node2*: 35
Total local states of node3: 34
Total number global states: 103
Possibly predicate: false
Running time cost: 484ms

Figure 9-14 test case 2 predicate 1 simulated bus possible predicate detection graphic result

9.4.2.1.1.4 *General result (Definitely predicate)*

Total local states of *node1*: 36
Total local states of *node2*: 35
Total local states of node3: 34
Total number global states: 103
Definitely predicate: false
Running time cost: 296ms

Figure 9-15 test case 2 predicate 1 simulated bus definitely predicate detection graphic result

9.4.2.1.2 Real system test result

9.4.2.1.2.1 *Local states of each node*

Vector time	Real time	State	Event
<0,0,0>	4.76E-03	11 11 11 11 11 0 0 0	internalEvent
<1,0,0>	0.101	12 12 12 0 0 0 0 0	internalEvent
<2,0,0>	0.101134	12 12 12 0 0 0 0 0	sendMsg
<3,3,3>	0.304042	11 11 11 11 11 0 0 0	receiveMsg
<4,3,3>	0.40399	12 12 12 0 0 0 0 0	internalEvent
<5,3,3>	0.404124	12 12 12 0 0 0 0 0	sendMsg
<6,6,6>	0.607042	11 11 11 11 11 0 0 0	receiveMsg
<7,6,6>	0.706938	12 12 12 0 0 0 0 0	internalEvent
<8,6,6>	0.707072	12 12 12 0 0 0 0 0	sendMsg
<9,9,9>	0.90995	11 11 11 11 11 0 0 0	receiveMsg
<10,9,9>	1.00999	12 12 12 0 0 0 0 0	internalEvent
<11,9,9>	1.010124	12 12 12 0 0 0 0 0	sendMsg

Table 9-19 node1 local states

Vector time	Real time	State	Event
<0,0,0>	0.003732	21 21 21 21 0 0 0 0	internalEvent
<2,1,0>	0.102022	22 22 22 0 0 0 0 0	receiveMsg
<2,2,0>	0.20227	21 21 21 21 0 0 0 0	internalEvent
<2,3,0>	0.202428	21 21 21 21 0 0 0 0	sendMsg
<5,4,3>	0.405028	22 22 22 0 0 0 0 0	receiveMsg
<5,5,3>	0.505278	21 21 21 21 0 0 0 0	internalEvent
<5,6,3>	0.505436	21 21 21 21 0 0 0 0	sendMsg
<8,7,6>	0.70804	22 22 22 0 0 0 0 0	receiveMsg
<8,8,6>	0.808287	21 21 21 21 0 0 0 0	internalEvent
<8,9,6>	0.808446	21 21 21 21 0 0 0 0	sendMsg
<11,10,9>	1.011046	22 22 22 0 0 0 0 0	receiveMsg

Table 9-20 node2 local states

Vector time	Real time	State	Event
<0,0,0>	0.004526	31 31 31 0 0 0 0 0	internalEvent
<2,3,1>	0.203034	32 32 32 32 0 0 0 0	receiveMsg
<2,3,2>	0.302982	31 31 31 0 0 0 0 0	internalEvent
<2,3,3>	0.303122	31 31 31 0 0 0 0 0	sendMsg
<5,6,4>	0.50644	32 32 32 32 0 0 0 0	receiveMsg
<5,6,5>	0.606068	31 31 31 0 0 0 0 0	internalEvent
<5,6,6>	0.606202	31 31 31 0 0 0 0 0	sendMsg
<8,9,7>	0.809514	32 32 32 32 0 0 0 0	receiveMsg
<8,9,8>	0.908938	31 31 31 0 0 0 0 0	internalEvent
<8,9,9>	0.909072	31 31 31 0 0 0 0 0	sendMsg
<11,12,10>	1.112464	32 32 32 32 0 0 0 0	receiveMsg

Table 9-21 node3 local states

9.4.2.1.2.2 *Consistent global states*

Global vector time	Local vector time	Real time	State	Event
<0,0,0>	<0,0,0>	4.76E-03	11 11 11 11 11 0 0 0	internalEvent
	<0,0,0>	3.73E-03	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	4.53E-03	31 31 31 0 0 0 0 0	internalEvent
<1,0,0>	<1,0,0>	1.01E-01	12 12 12 0 0 0 0 0	internalEvent
	<0,0,0>	3.73E-03	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	4.53E-03	31 31 31 0 0 0 0 0	internalEvent
<2,0,0>	<2,0,0>	0.101134	12 12 12 0 0 0 0 0	sendMsg
	<0,0,0>	3.73E-03	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	4.53E-03	31 31 31 0 0 0 0 0	internalEvent
<2,1,0>	<2,0,0>	0.101134	12 12 12 0 0 0 0 0	sendMsg
	<2,1,0>	1.02E-01	22 22 22 0 0 0 0 0	receiveMsg
	<0,0,0>	4.53E-03	31 31 31 0 0 0 0 0	internalEvent
<2,2,0>	<2,0,0>	0.101134	12 12 12 0 0 0 0 0	sendMsg
	<2,2,0>	0.20227	21 21 21 21 0 0 0 0	internalEvent
	<0,0,0>	4.53E-03	31 31 31 0 0 0 0 0	internalEvent

Table 9-22 global states

9.4.2.1.2.3 *General result (possibly predicate)*

Total local states of *node1*: 469
Total local states of *node2*: 469
Total local states of node3: 469
Total number global states: 1405
Possibly predicate: false
Running time cost: 94938ms

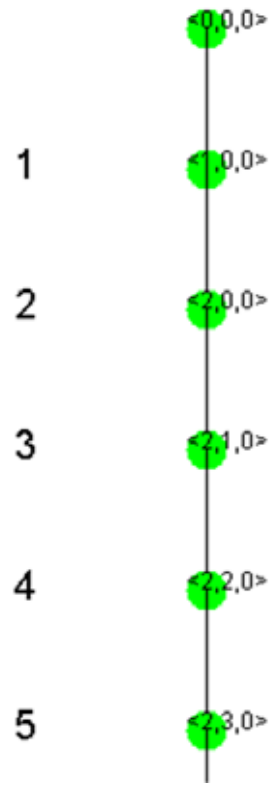


Figure 9-16 test case 2 predicate 1 real bus possibly predicate detection graphic result

9.4.2.1.2.4 *General result (Definitely predicate)*

Total local states of *node1*: 469
Total local states of *node2*: 469
Total local states of node3: 469
Total number global states: 1405
Definitely predicate: false
Running time cost: 106266ms

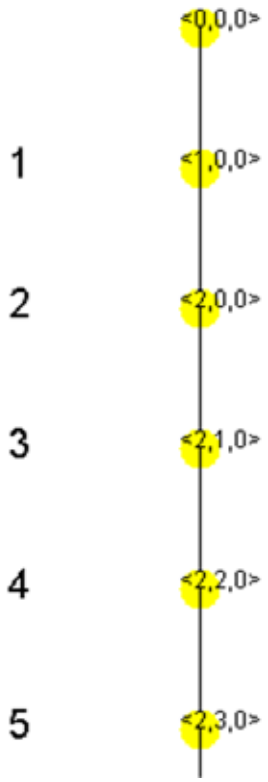


Figure 9-17 test case 2 predicate 1 real bus definitely predicate detection graphic result

9.4.3 Result analysis

The execution of this text case is shown in Figure 9-18. This test case is analysed with the same method as the test case 1.

1. To verify the function that assigns vector clocks using Table 9-19 node1 local states, Table 9-20 node2 local states, and Table 9-21 node3 local states against the execution Figure 9-18. The result of the verification is success.

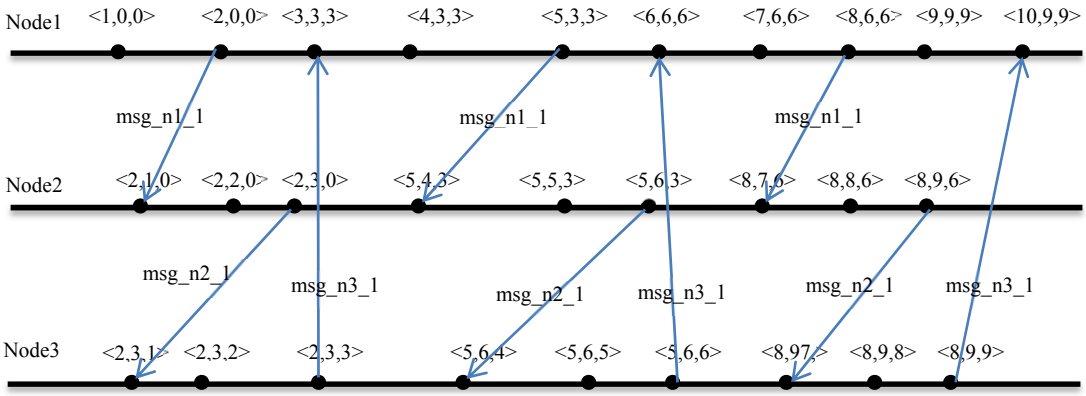


Figure 9-18 test case 2 execution

2. To verify the function that evaluates consistent global state. Going through Table 9-22, using algorithm in section 7.2.2 manually verifying consistent global states that are found by prototype. The result of the verification is success.
3. Validate the result of predicate evaluated by the prototype. The result of the validation is successes.
4. Visually checking the lattice diagram.

All steps also are applied on the real system. The same results are obtained.

9.5 Test case 3

Test case 3 tests the effect of the environment variable in the state machine. Environment variables are data objects global to the CANoe environment, and are used to link the functions of a CANoe panel to CAPL programs (Vector 2004, p28).

9.5.1 Model explanation

9.5.1.1 State machine diagram

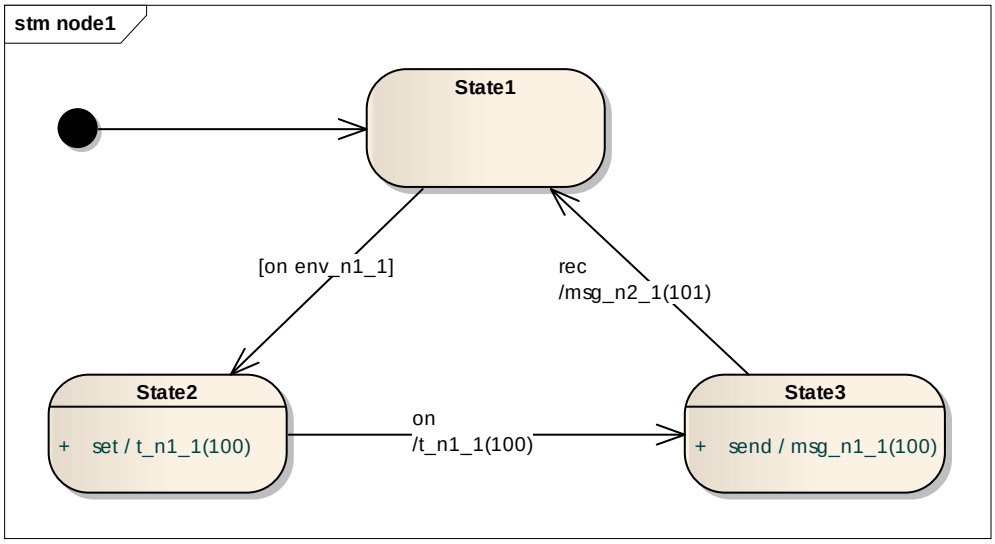


Figure 9-19 test case 3 state machine 1

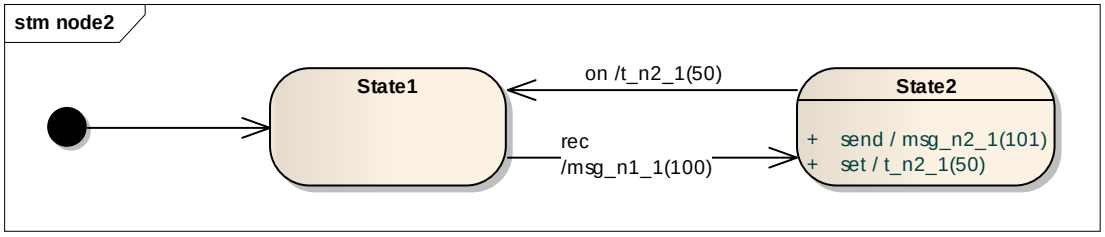


Figure 9-20 test case 3 state machine 2

9.5.1.2 Communication matrix

MessageID	SendNodeNum	receive: nodeNum
100	1	2
101	2	1

Table 9-23 test case 3 communication matrix

9.5.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	1	1	1	1	11	0	1	1	0	0
2	0	2	2	2	2	0	2	0	0	0
3	3	3	3	3	3	3	3	3	0	0

Table 9-24 test case 3 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	21	21	21	21	21	21	21	21	0	0
2	22	22	22	22	22	22	22	0	0	0

Table 9-25 test case 3 *node2* local states

9.5.2 Test different inputs

9.5.2.1 Predicate 1

node2.var1>*node1*.var2 || *node1*.var2<*node2*.var5

9.5.2.1.1 Simulated system test result

9.5.2.1.1.1 *Local states of each node*

Vector time	Real time	State	Event
<0,0>	2.44E-04	1 1 1 1 11 0 1 1	internalEvent
<1,0>	4.92E-04	0 2 2 2 2 0 2 0	internalEvent
<2,0>	0.100242	3 3 3 3 3 3 3 3	internalEvent
<3,0>	0.100378	3 3 3 3 3 3 3 3	sendMsg
<4,3>	0.150982	1 1 1 1 11 0 1 1	receiveMsg
<5,3>	1.142855	0 2 2 2 2 0 2 0	internalEvent
<6,3>	1.242849	3 3 3 3 3 3 3 3	internalEvent
<7,3>	1.242985	3 3 3 3 3 3 3 3	sendMsg
<8,6>	1.293589	1 1 1 1 11 0 1 1	receiveMsg
<9,6>	1.437584	0 2 2 2 2 0 2 0	internalEvent

Table 9-26 node1 local states

Vector time	Real time	State	Event
<0,0>	7.22E-04	21 21 21 21 21 21 21 21	internalEvent
<3,1>	0.100612	22 22 22 22 22 22 22 0	receiveMsg
<3,2>	0.150608	21 21 21 21 21 21 21 21	internalEvent
<3,3>	0.150738	21 21 21 21 21 21 21 21	sendMsg
<7,4>	1.243219	22 22 22 22 22 22 22 0	receiveMsg
<7,5>	1.293215	21 21 21 21 21 21 21 21	internalEvent
<7,6>	1.293345	21 21 21 21 21 21 21 21	sendMsg

Table 9-27 node2 local states

9.5.2.1.1.2 *Consistent global states*

Global vector time	Local vector time	Real time	State	Event
<0,0>	<0,0>	2.44E-04	1 1 1 1 11 0 1 1	internalEvent
	<0,0>	7.22E-04	21 21 21 21 21 21 21 21	internalEvent
<1,0>	<1,0>	4.92E-04	0 2 2 2 2 0 2 0	internalEvent
	<0,0>	7.22E-04	21 21 21 21 21 21 21 21	internalEvent
<2,0>	<2,0>	0.100242	3 3 3 3 3 3 3 3	internalEvent
	<0,0>	7.22E-04	21 21 21 21 21 21 21 21	internalEvent
<3,0>	<3,0>	0.100378	3 3 3 3 3 3 3 3	sendMsg
	<0,0>	7.22E-04	21 21 21 21 21 21 21 21	internalEvent
<3,1>	<3,0>	0.100378	3 3 3 3 3 3 3 3	sendMsg
	<3,1>	0.100612	22 22 22 22 22 22 22 0	receiveMsg
<3,2>	<3,0>	0.100378	3 3 3 3 3 3 3 3	sendMsg
	<3,2>	0.150608	21 21 21 21 21 21 21 21	internalEvent
<3,3>	<3,0>	0.100378	3 3 3 3 3 3 3 3	sendMsg
	<3,3>	0.150738	21 21 21 21 21 21 21 21	sendMsg
<4,3>	<4,3>	0.150982	1 1 1 1 11 0 1 1	receiveMsg
	<3,3>	0.150738	21 21 21 21 21 21 21 21	sendMsg
<5,3>	<5,3>	1.142855	0 2 2 2 2 0 2 0	internalEvent
	<3,3>	0.150738	21 21 21 21 21 21 21 21	sendMsg

Table 9-28 global states

9.5.2.1.1.3 *General result (possibly predicate)*

Total local states of *node1*: 101
Total local states of *node2*: 76
Total number global states: 176
Possibly predicate: true
Running time cost: 506ms

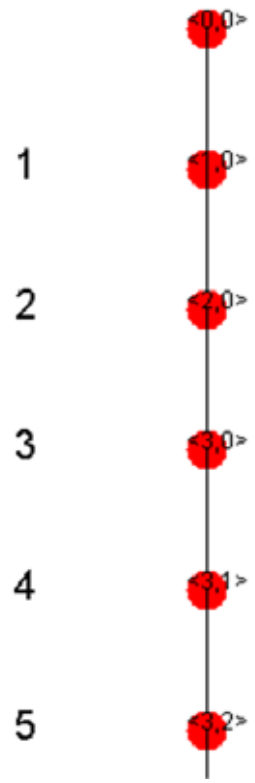


Figure 9-21 test case 3 predicate 1 simulated bus possibly predicate detection graphic result

9.5.2.1.1.4 *General result (Definitely predicate)*

Total local states of *node1*: 101
Total local states of *node2*: 76
Total number global states: 176
Definitely predicate: true
Running time cost: 369ms

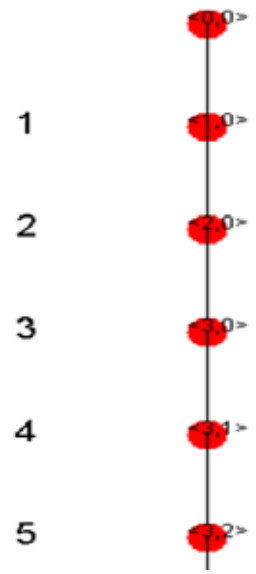


Figure 9-22 test case 3 predicate 1 simulated bus definitely predicate detection graphic result

9.5.2.1.2 Real system test result

9.5.2.1.2.1 *General result (possibly predicate)*

Total local states of *node1*: 597
Total local states of *node2*: 895
Total number global states: 1493
Possibly predicate: true
Running time cost: 1703ms

9.5.2.1.2.2 *General result (Definitely predicate)*

Total local states of *node1*: 597
Total local states of *node2*: 895
Total number global states: 1493
Definitely predicate: true
Running time cost: 1342ms

9.5.3 Result analysis

The prototype is verified and validated for this test case as well as test case 1 and test case 2.

1. To verify the function that assigns vector clocks using Table 9-26 node1 local states and Table 9-27 node2 local states against the execution Figure 9-23 test case 3 execution. The result of the verification is success.
2. To verify the function that evaluates consistent global state. Going through Table 9-28, using algorithm in section 7.2.2 manually verifying consistent global states that are found by prototype. The result of the verification is success.
3. Validate the result of predicate evaluated by the prototype. The result of the validation is successes.
4. Visually checking the lattice diagram.

All steps also are applied on the real system. The same results are obtained.

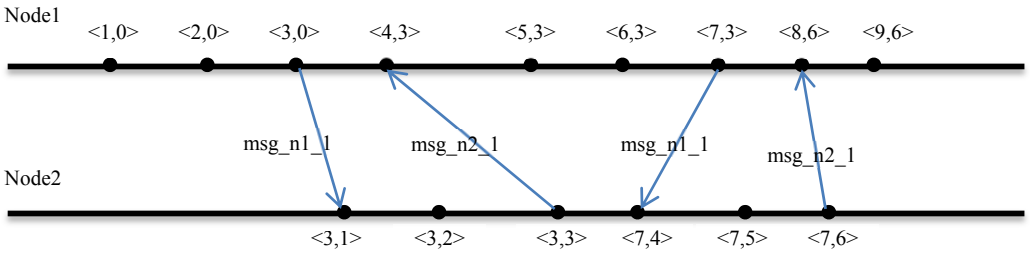


Figure 9-23 test case 3 execution

9.6 Test case 4

This test case is randomly generated. *node1* only sends messages. The transition routes of the other nodes depend on these messages. The possibility of different execution traces should be high.

9.6.1 Model explanation

9.6.1.1 State machine diagram

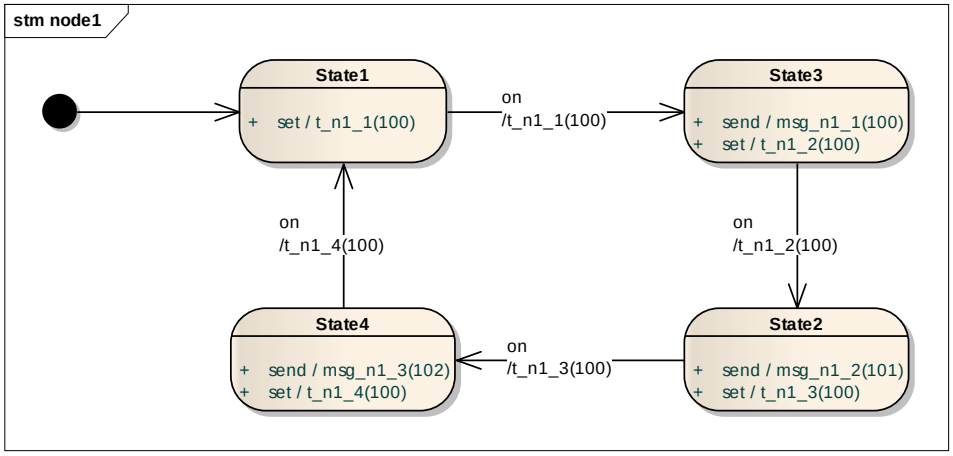


Figure 9-24 test case 4 state machine 1

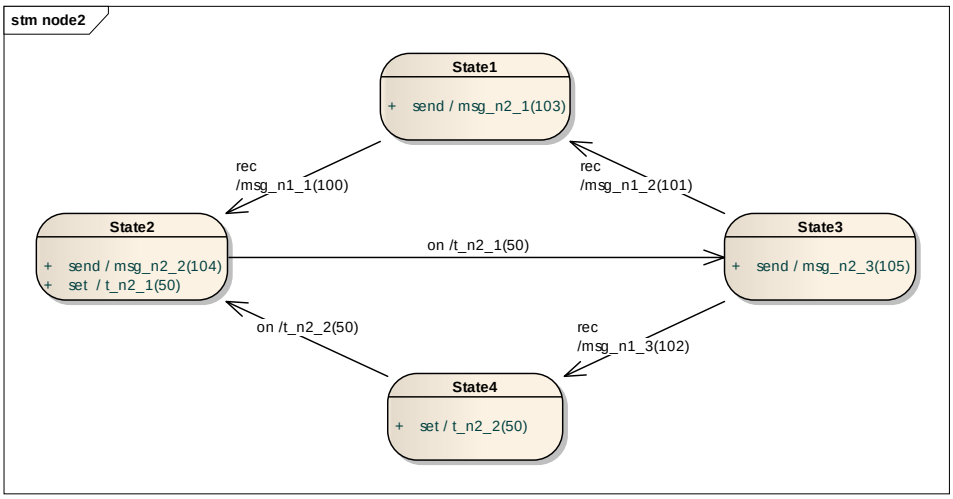


Figure 9-25 test case 4 state machine 2

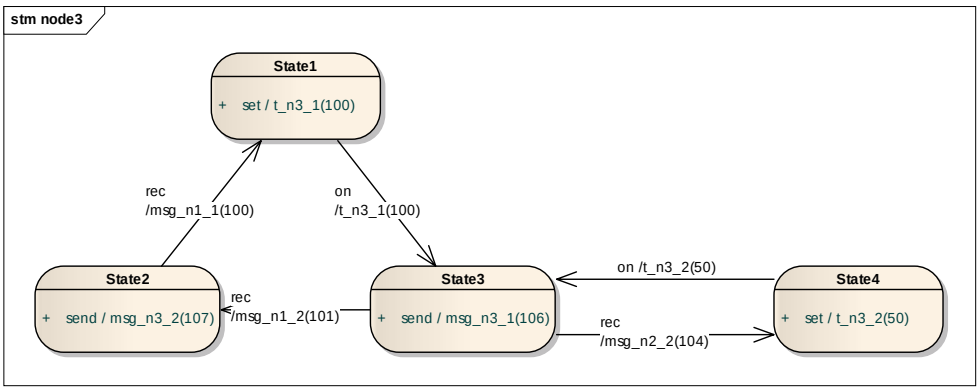


Figure 9-26 test case 4 state machine 3

9.6.1.2 Communication matrix

MessageID	SendNodeNum	receive: nodeNum
100	1	2
100	1	3
101	1	2
101	1	3
102	1	2
103	2	
104	2	3
105	2	
106	3	
107	3	

Table 9-29 test case 4 communication matrix

9.6.1.3 Local states

stateNum	var1	var2	0	var4	var5	var6	var7	var8	var9	var10
1	11	11	11	11	11	0	0	0	0	0
2	12	12	12	12	12	2	0	0	0	0
3	13	13	13	13	13	0	0	0	0	0
4	14	14	14	14	14	14	0	0	0	0

Table 9-30 test case 4 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	21	21	21	21	21	0	0	0	0	0
2	22	22	22	22	22	0	0	0	0	0
3	23	23	23	23	23	0	0	0	0	0
4	24	24	24	24	24	0	0	0	0	0

Table 9-31 test case 4 *node2* local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	31	31	31	31	31	0	0	0	0	0
2	32	32	32	32	32	32	0	0	0	0
3	33	33	33	33	33	33	0	0	0	0
4	34	34	34	34	34	34	0	0	0	0

Table 9-32 test case 4 node 3 local states

9.6.2 Test different inputs

9.6.2.1 Predicate 1

node2.var2>21 && *node1*.var2==11

9.6.2.1.1 simulated system test result

9.6.2.1.1.1 *General result (possibly predicate)*

Total local states of *node1*: 10
Total local states of *node2*: 13
Total local states of node3: 8
Total number global states: 492
Possibly predicate: true
Running time cost: 345ms

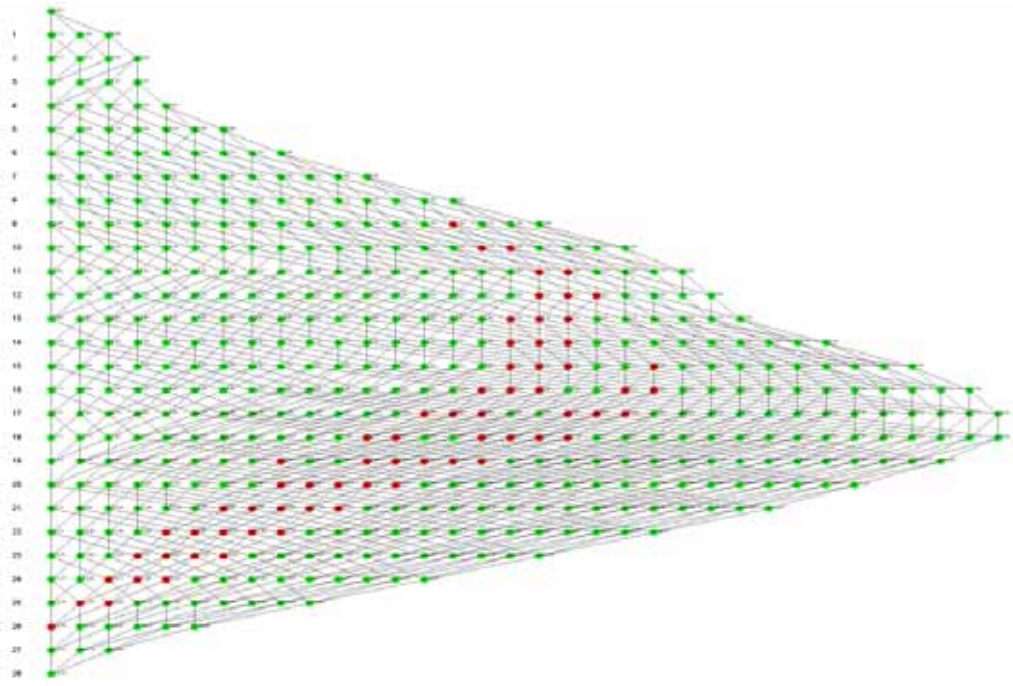


Figure 9-27 test case 4 predicate 1 simulated bus possibly predicate detection graphic result

9.6.2.1.1.2 *General result (Definitely predicate)*

Total local states of *node1*: 10
Total local states of *node2*: 13
Total local states of node3: 8
Total number global states: 492
Definitely predicate: false
Running time cost: 331ms

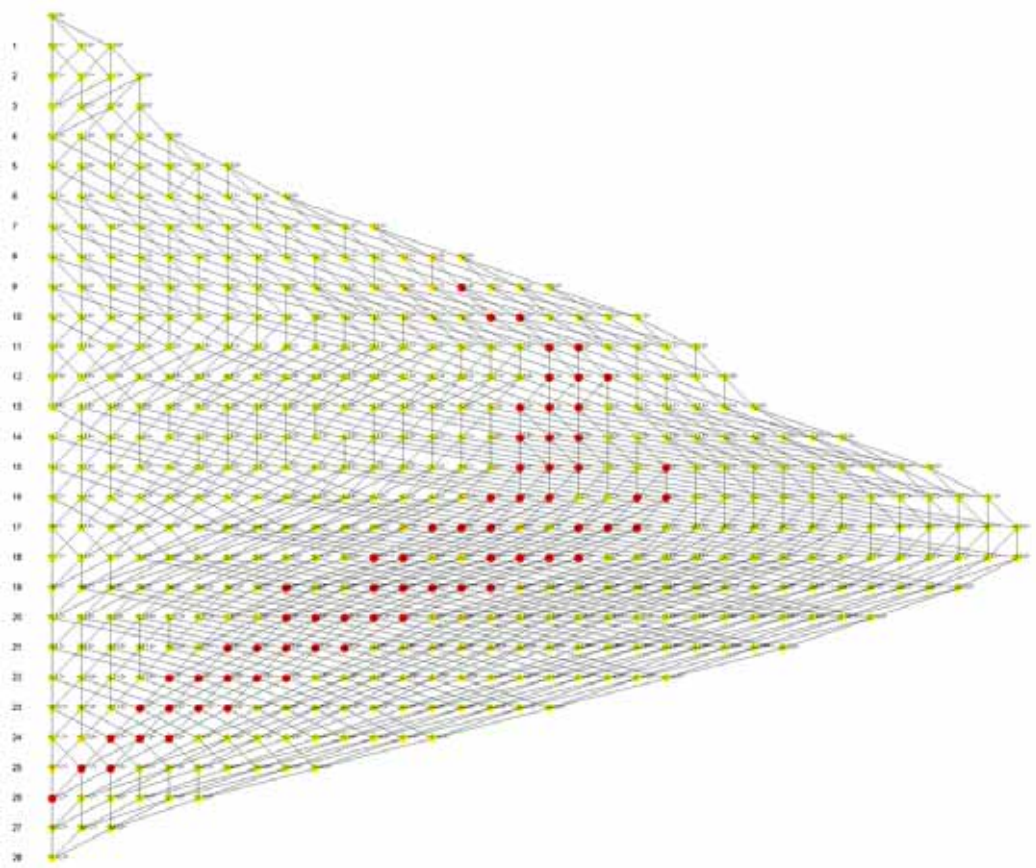


Figure 9-28 test case 4 predicate 1 simulated bus definitely predicate detection graphic result

9.6.2.1.2 Real system test result

9.6.2.1.2.1 *General result (possibly predicate)*

Real time period of the log file: 0.001696-0.901011
Total local states of *node1*: 16
Total local states of *node2*: 14
Total local states of node3: 10
Total number global states: 833
Possibly predicate: true
Running time cost: 641ms

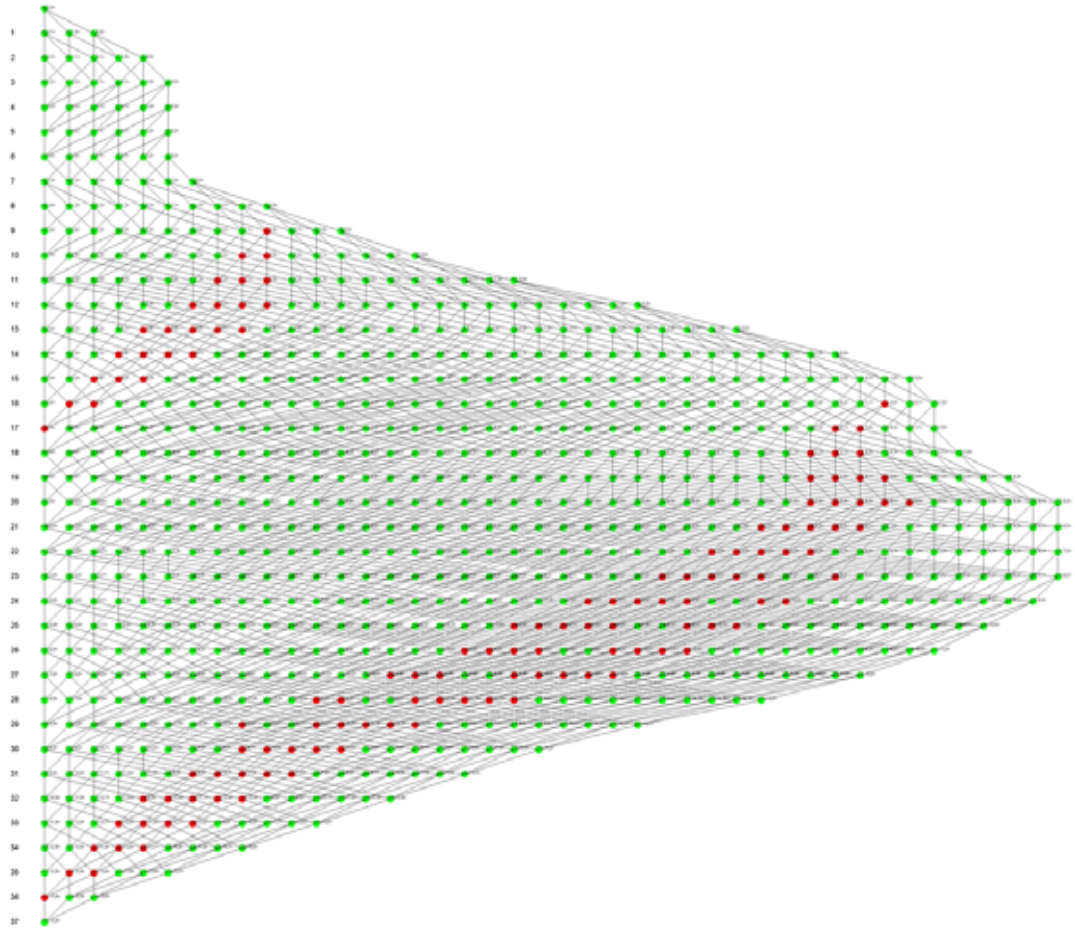


Figure 9-29 test case 4 predicate 1 real bus possibly predicate detection graphic result

9.6.2.1.2.2 *General result (Definitely predicate)*

Total local states of *node1*: 16
Total local states of *node2*: 14
Total local states of node3: 10
Total number global states: 833
Definitely predicate: false
Running time cost: 482ms

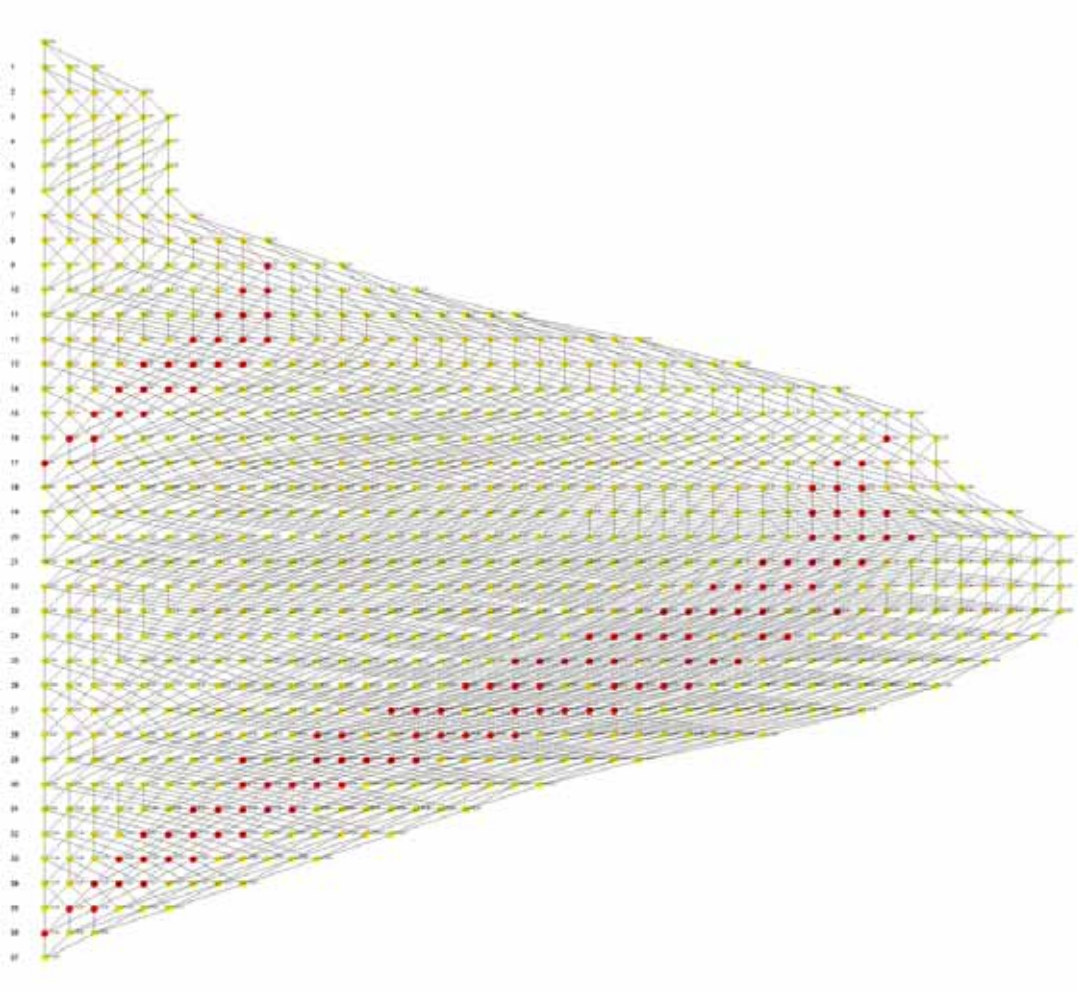


Figure 9-30 test case 4 predicate 1 real bus definitely predicate detection graphic result

9.6.2.2 Predicate 2

Predicate expression:

node1.var2>21 && node3.var2>=11 || *node1*.var6<=11 &&

node3.var2==*node2*.var2 && *node2*.var2>node3.var7 &&

node1.var2!=*node2*.var5 || node3.var5>*node1*.var7 && *node2*.var2==node3.var3

9.6.2.2.1 simulated system test result

9.6.2.2.1.1 *General result (possibly predicate)*

Real time period of the log file: 0.000236- 0.700626
Total local states of node1: 14
Total local states of node2: 16
Total local states of node3: 13
Total number global states: 1097
Possibly predicate: true
Running time cost: 515ms

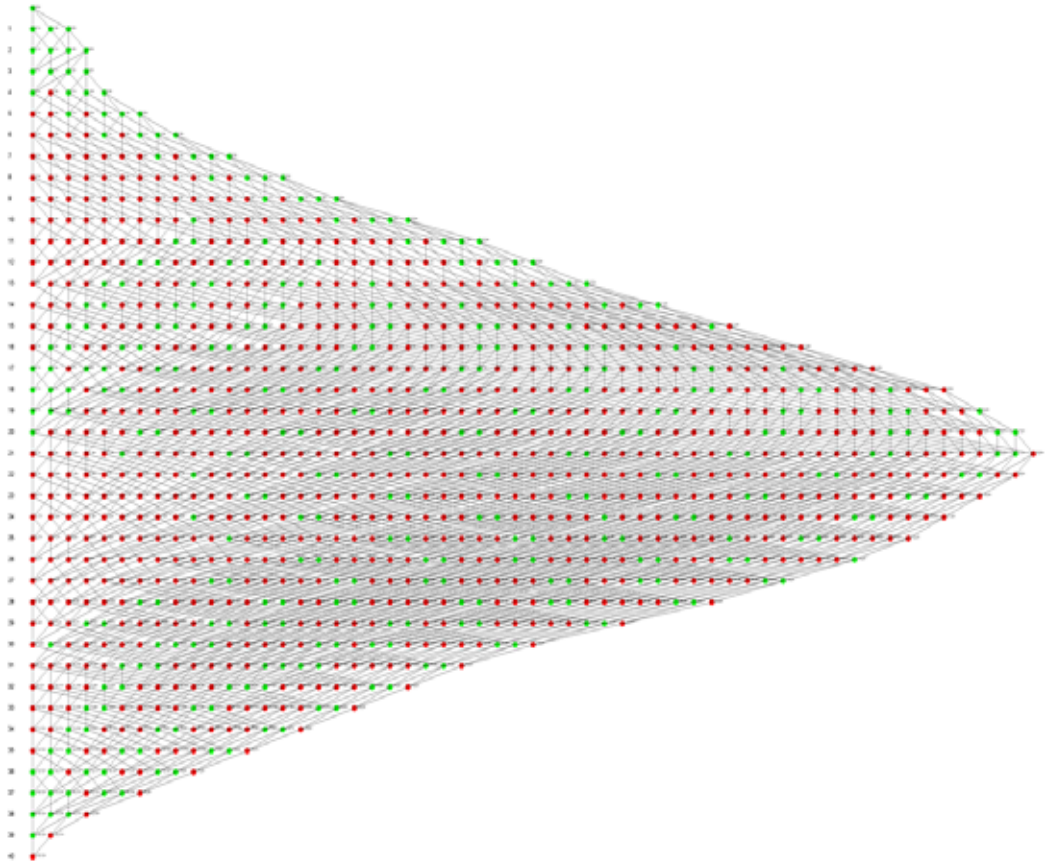


Figure 9-31 test case 4 predicate 2 simulated bus possible predicate detection graphic result

9.6.2.2.1.2 *General result (Definitely predicate)*

Total local states of node1: 14
Total local states of node2: 16
Total local states of node3: 13
Total number global states: 1097
Definitely predicate: true
Running time cost: 484ms

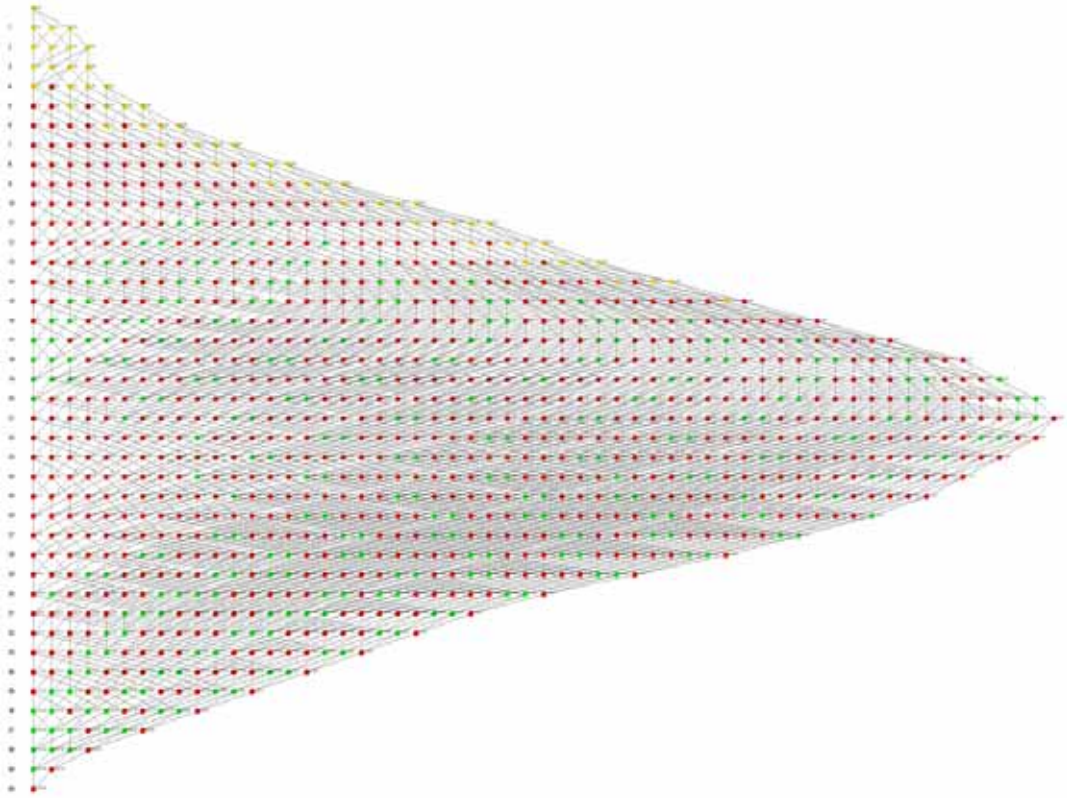


Figure 9-32 test case 4 predicate 2 simulated bus definitely predicate detection graphic result

9.6.2.2.2 Real system test result

9.6.2.2.2.1 *General result (possibly predicate)*

Real time period of the log file: 0-0.701443
Total local states of node1: 14
Total local states of node2: 9
Total local states of node3: 10
Total number global states: 381
Possibly predicate: true
Running time cost: 312ms

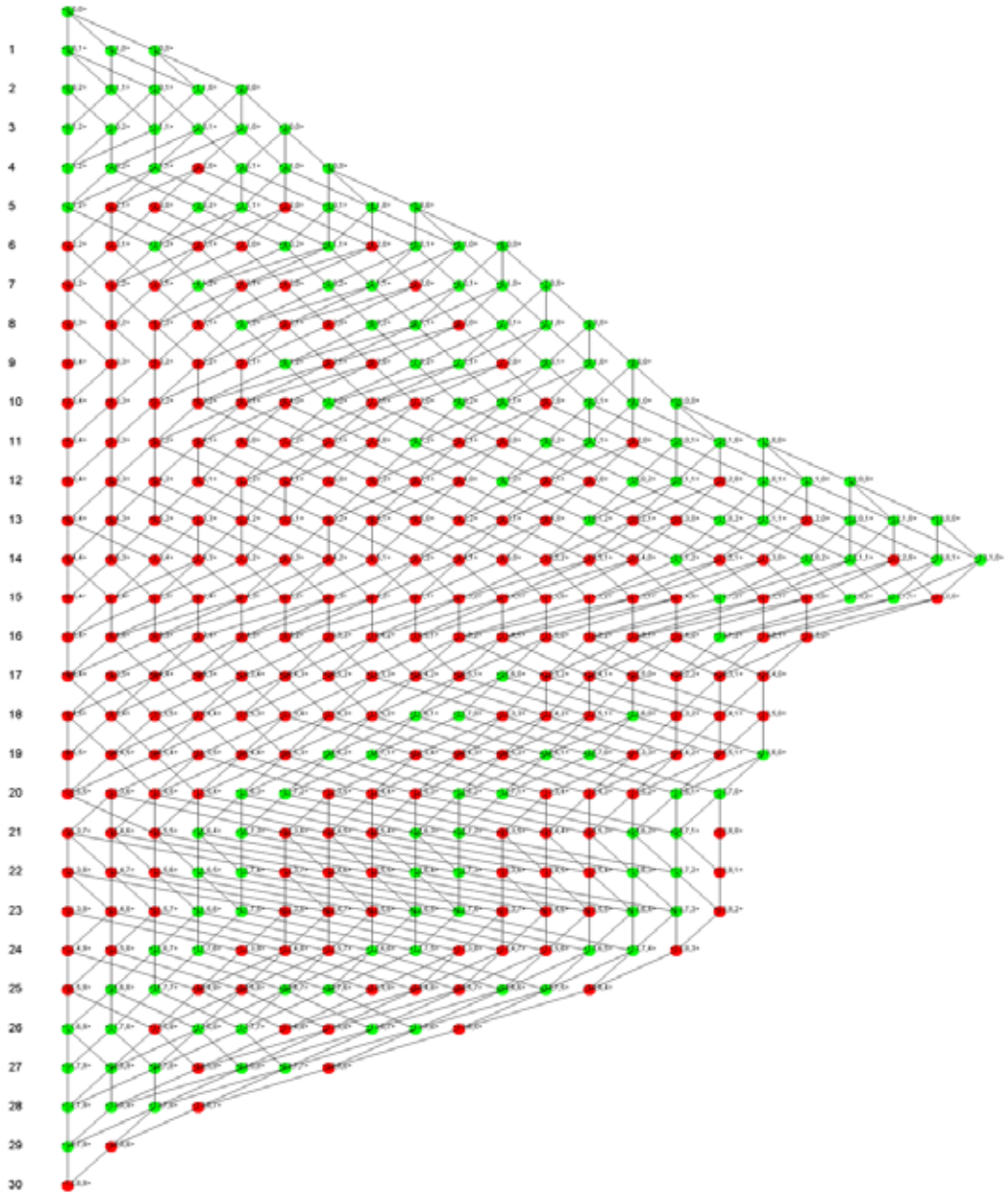


Figure 9-33 test case 4 predicate 2 real bus possible predicate detection graphic result

9.6.2.2.2.2 *General result (Definitely predicate)*

Total local states of node1: 14
Total local states of node2: 9
Total local states of node3: 10
Total number global states: 381
Definitely predicate: true
Running time cost: 328ms

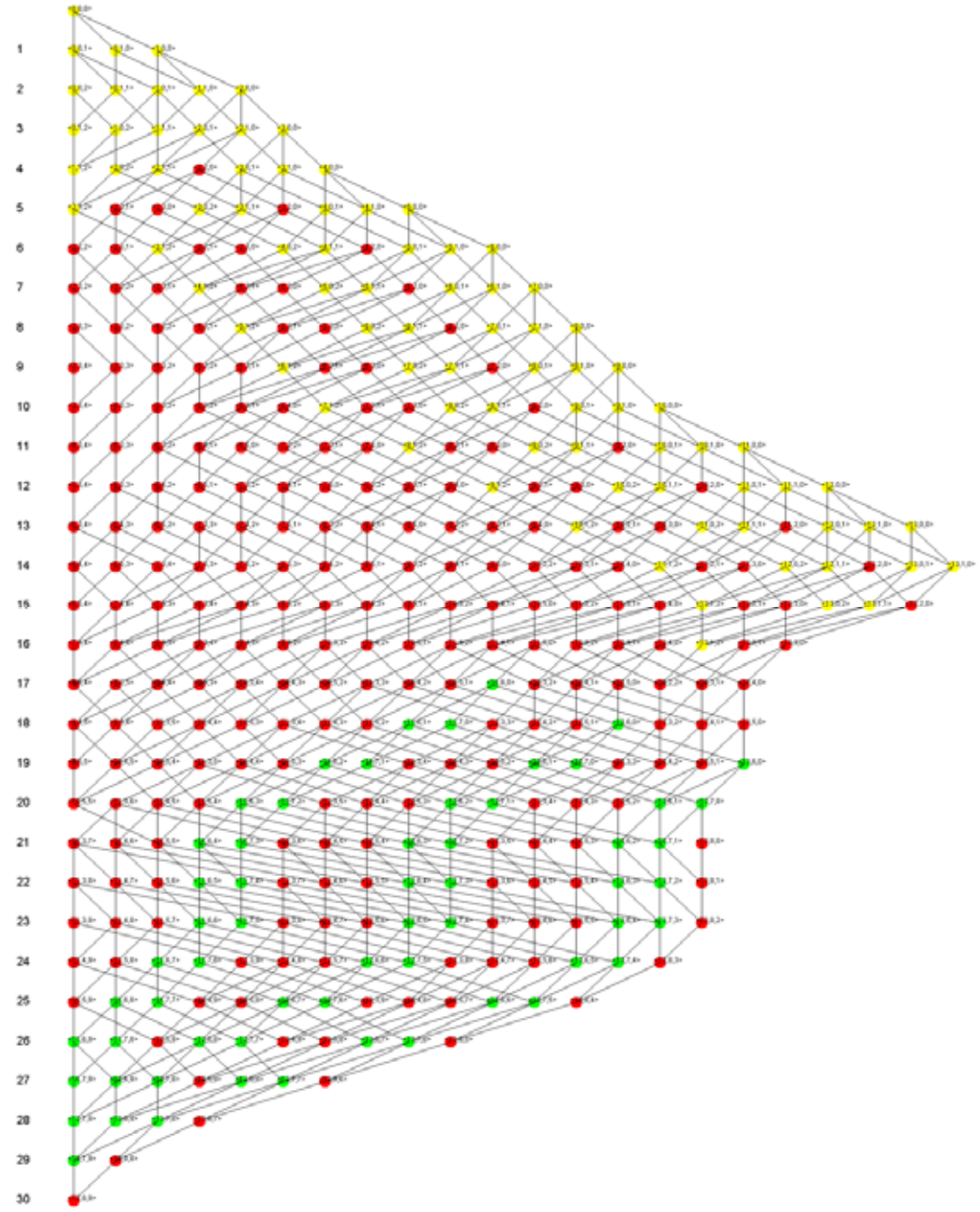


Figure 9-34 test case 4 predicate 2 real bus definitely predicate detection graphic result

9.6.3 Result analysis

Due to this test case being generated randomly (previous test cases are designed), the execution track is arbitrary. The method to analyze this test case is a little different from the previous method. In the previous test cases, the execution diagram which is used to verify the function assigning the vector clocks is manually generated from the state diagram. But in this test case the execution diagram is manually generated from the result of the prototype vector clock assigning function. Using this execution diagram against the state diagram verifies the vector clock assigning function. If the execution logically matches the state diagram, then the function is verified. Otherwise it is not verified.

The execution diagram generated from the simulated system is different from the diagram generated from real system. Two diagrams are illustrated in Figure 9-35 and Figure 9-36. The real system needs to start the node running on another machine manually. It takes a longer delay than the simulated system. The simulated system almost starts all nodes at the same time.

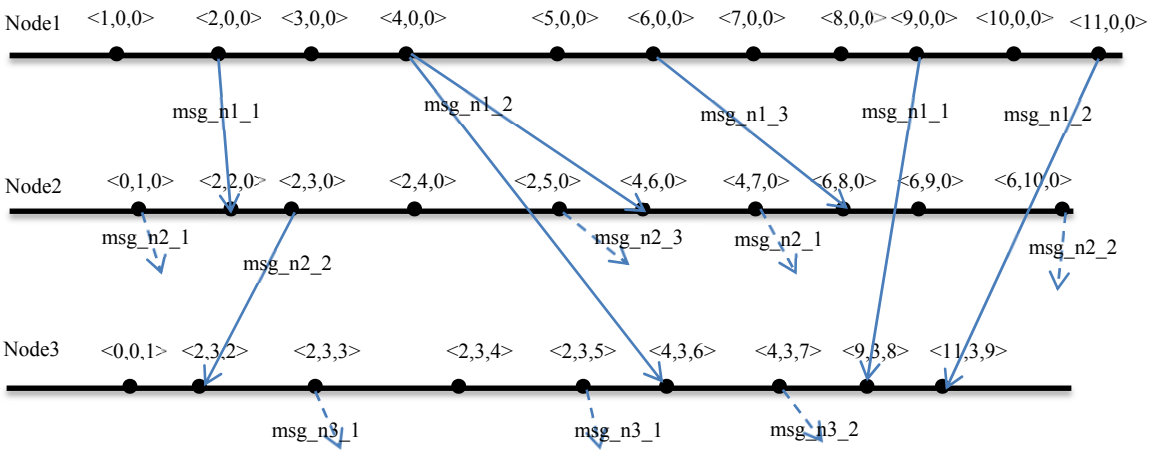


Figure 9-35 simulated system execution

As shown in Figure 9-35, the sending event with vector time <4,0,0> in node1has two arrows. These two arrows do not mean that two messages are sent. It is one message (in CAN network, all messages are broadcast), two other nodes filter the message. It can be that any number of nodes filters this message, so the sending event vector clock does not increase.

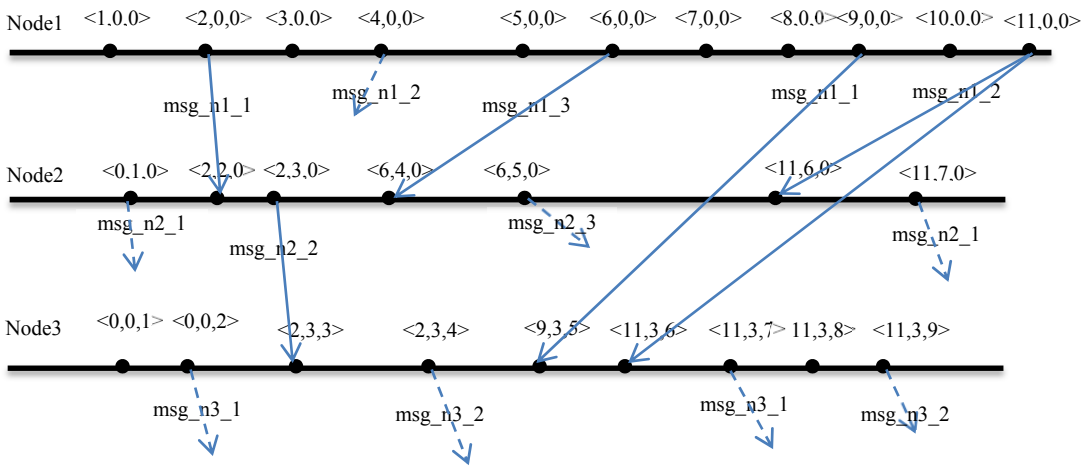


Figure 9-36 real system execution

Using the execution diagrams against the state machine diagrams verify vector times in the execution diagram. To verify the vector time need to manually evaluate if the vector times match the state diagram. The evaluation needs to consider the causality of events. These considerations are:

1. The transition caused by receiving event should happen after the sending event.
2. Two states on two ends of the transition in the state machine diagram have to match two corresponding events and transition on the execution diagram. E.g. in Figure 9-24 between *state1* and *state2* is transition *t_n1_1*, *state1* move to *state2* has to be when the timer *t_n1_1* expired. Otherwise the state move is not valid.

After evaluating these two execution diagrams, they are all validated. So the function that assigns vector times is verified.

The remaining steps are to verify the function evaluating consistent global state, to validate the predicate evaluation result, and validate the lattice.

The above four test cases verify and validate the prototype. The next few test cases will test the performance of the prototype.

9.7 Test case 5

This test case consists of 4 state machines. One of the state machines contains an environment variable. The change of the environment variable may cause the transition to a different execution state.

9.7.1 Model explanation

9.7.1.1 State machine diagram

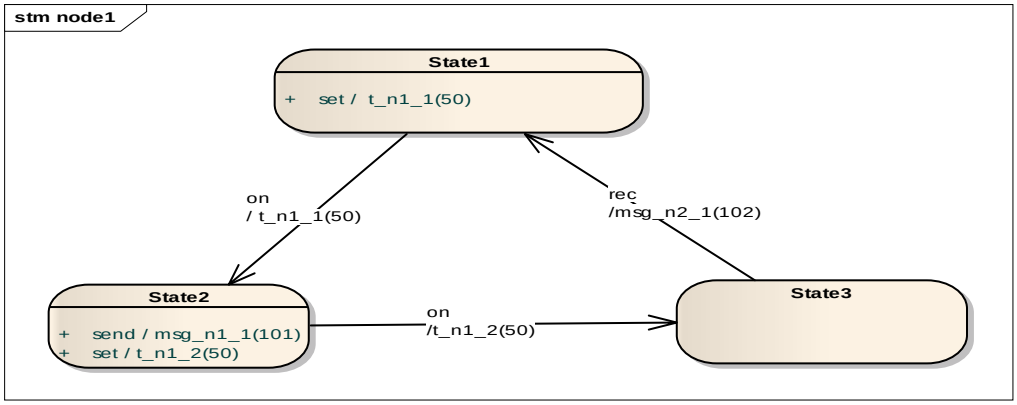


Figure 9-37 test case 5 state machine 1

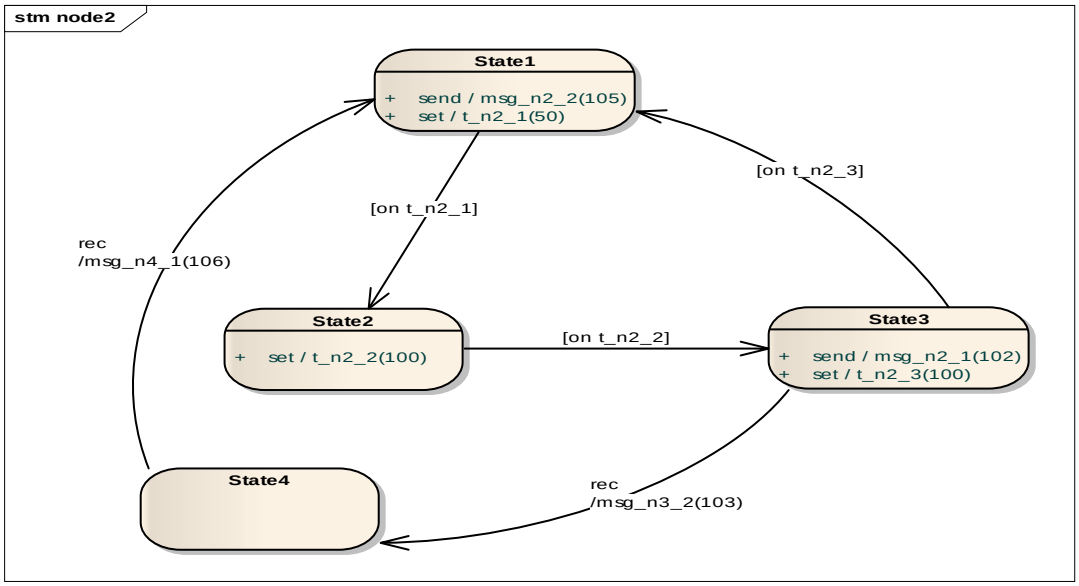


Figure 9-38 test case 5 state machine 2

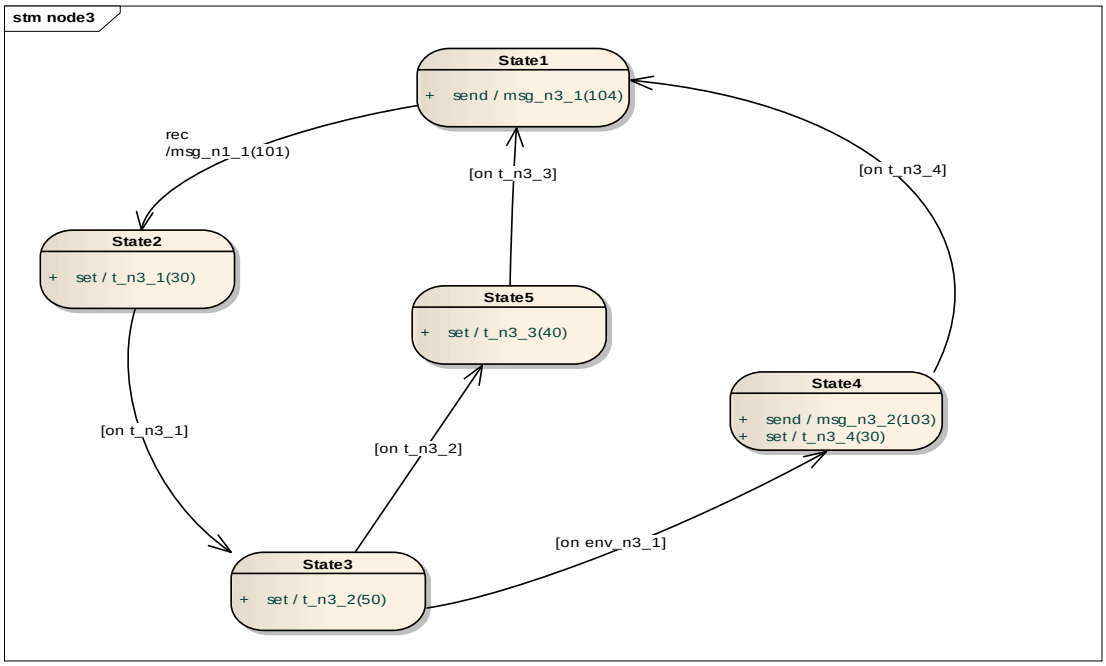


Figure 9-39 test case 5 state machine 3

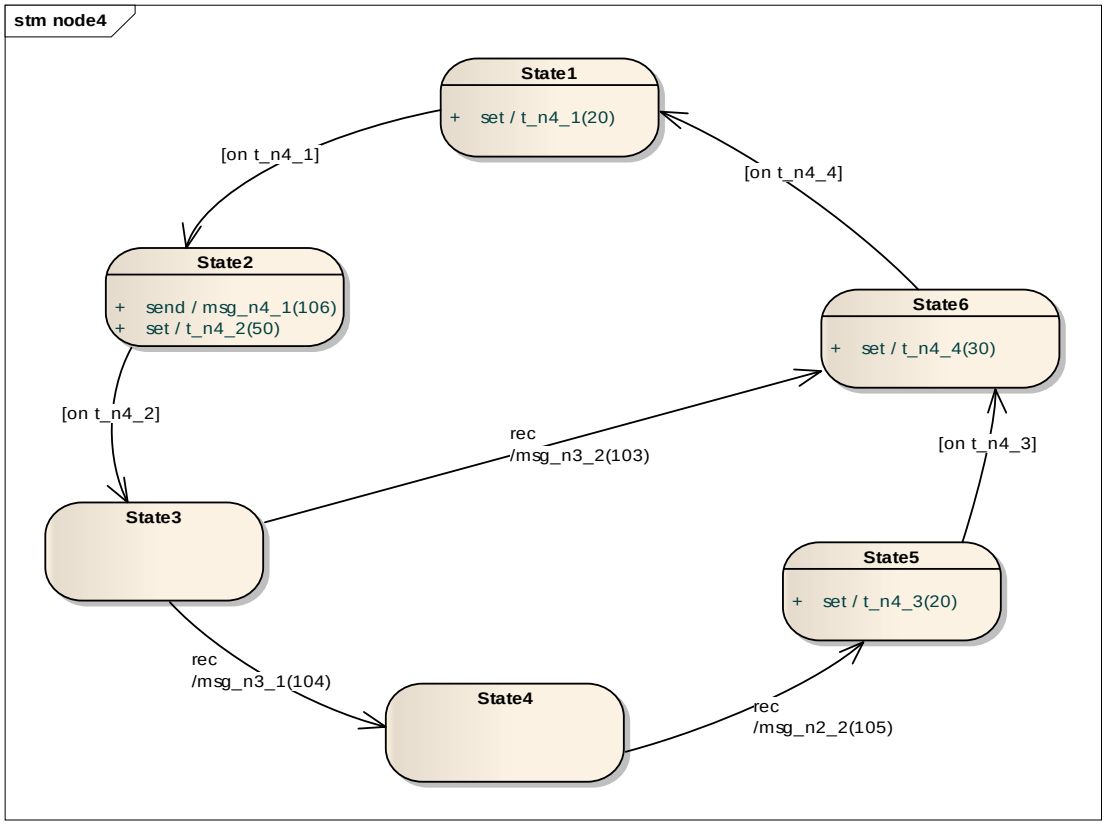


Figure 9-40 test case 5 state machine 4

9.7.1.2 Communication matrix

MessageID	SendNodeNum	receive: nodeNum
101	1	3
102	2	1
105	2	4
104	3	4
103	3	2
103	3	4
106	4	2

Table 9-33 test case 5 communication matrix

9.7.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	11	11	11	11	11	0	0	0	0
2	12	12	12	12	12	12	12	12	0	0
3	13	13	13	13	13	13	0	0	0	0

Table 9-34 test case 5 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	21	21	21	21	21	21	21	0	0	0
2	22	22	22	22	22	22	22	0	0	0
3	23	23	23	23	23	23	23	0	0	0
4	24	24	24	24	24	24	24	24	0	0

Table 9-35 test case 5 *node2* local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	31	31	31	31	31	31	31	0	0	0
2	32	32	32	32	32	32	32	32	0	0
3	33	33	33	33	33	33	33	0	0	0
4	34	34	34	34	34	34	34	0	0	0
5	35	35	35	35	35	35	35	35	0	0

Table 9-36 test case 5 node 3 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	41	41	41	41	41	41	41	41	0	0
2	42	42	42	42	42	42	42	42	0	0
3	43	43	43	43	43	43	43	43	43	0
4	44	44	44	44	44	44	44	44	0	0
5	45	45	45	45	45	45	45	45	45	0
6	46	46	46	46	46	46	46	46	0	0

Table 9-37 test case 5 node 4 local states

9.7.2 Test different inputs

9.7.2.1 Predicate 1

node3.var1<*node2*.var2 || node4.var2<*node1*.var7 || node4.var7<= 0 &&

node2.var4==*node1*.var6

9.7.2.1.1 Simulated system test result

9.7.2.1.1.1 *General result (possibly predicate)*

Real time period of the log file: 2.370622- 3.950638
Total local states of *node1*: 10
Total local states of *node2*: 12
Total local states of node3: 10
Total local states of node4: 9
Total number global states: 454
Possibly predicate: false
Running time cost: 383ms

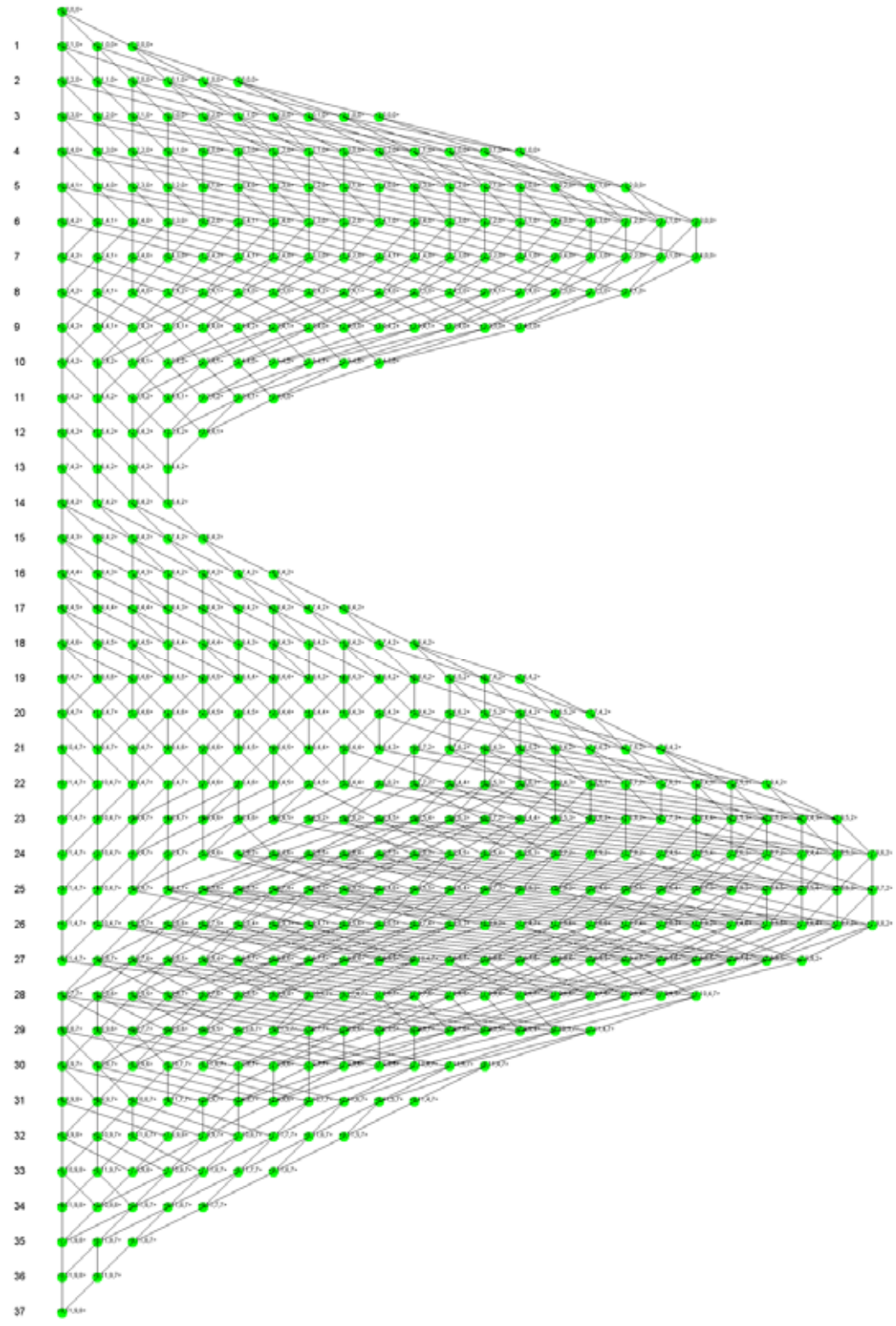


Figure 9-41 test case 5 predicate 1 simulated bus possibly predicate detection graphic result

9.7.2.1.1.2 *General result (Definitely predicate)*

Real time period of the log file: 0.000000- 0.400392
Total local states of *node1*: 8
Total local states of *node2*: 10
Total local states of node3: 12
Total local states of node4: 12
Total number global states: 566
Possibly predicate: false
Running time cost: 473ms

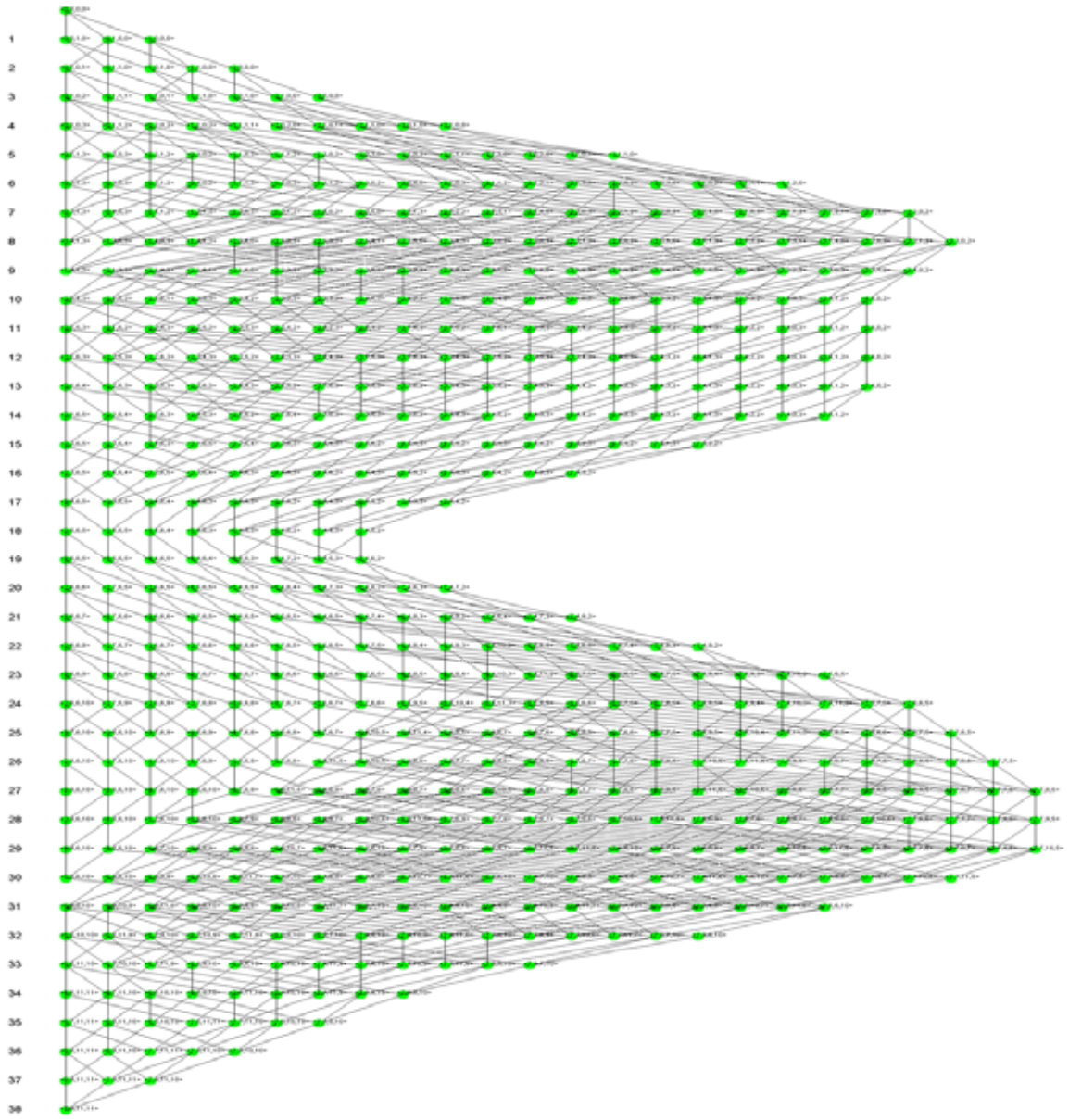


Figure 9-42 test case 5 predicate 1 simulated bus definitely predicate detection graphic result

9.7.2.1.1.3 *General result*

Real time period of the log file: 0.231044- 0.701058
Total local states of node1: 8
Total local states of node2: 10
Total local states of node3: 9
Total local states of node4: 8
Total number global states: 236
Possibly predicate: false
Running time cost: 424ms

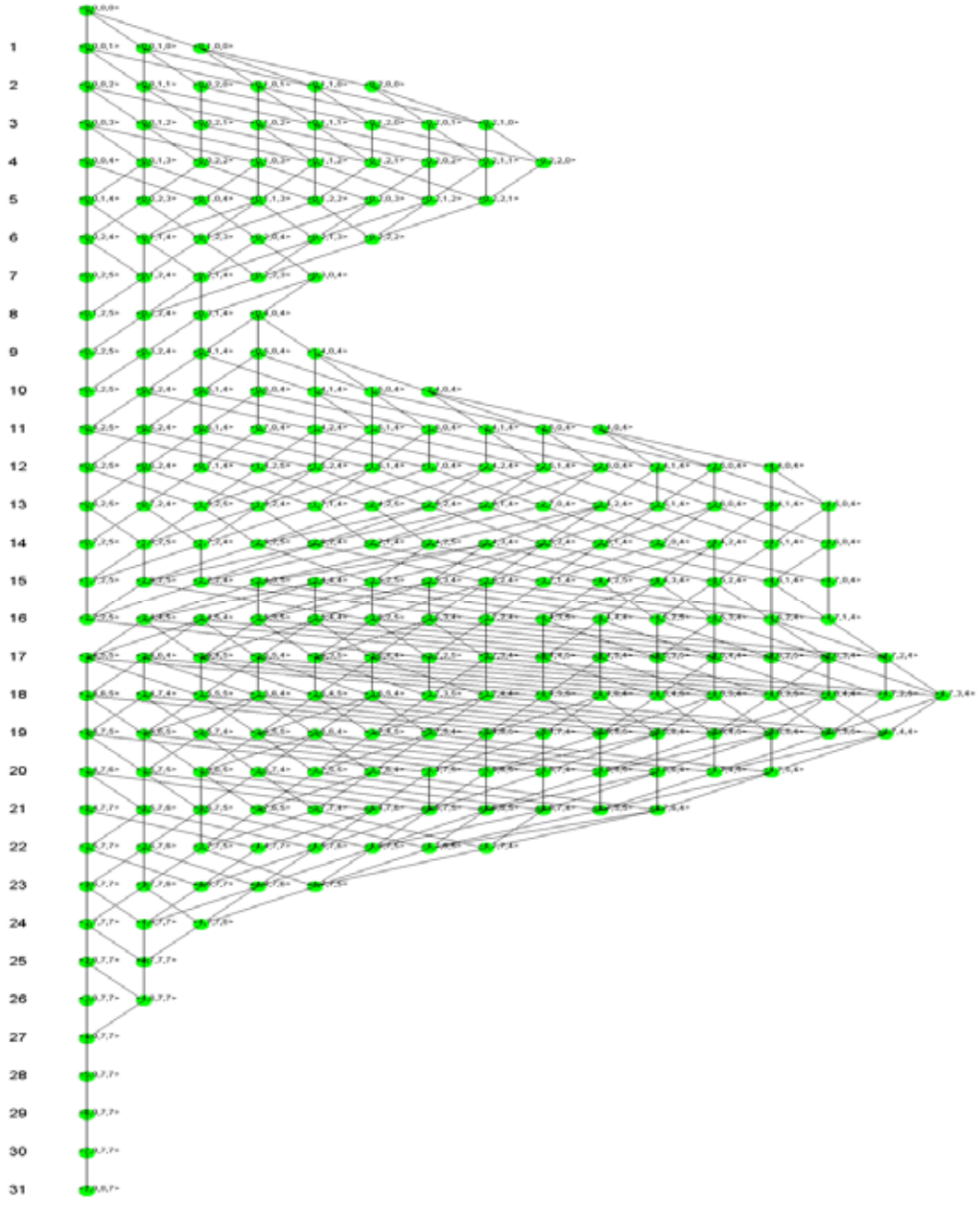


Figure 9-43 graphic result

9.7.2.1.1.4 *General result*

Real time period of the log file: 0.000000- 0.400392
Total local states of node1: 12
Total local states of node2: 14
Total local states of node3: 12
Total local states of node4: 12
Total number global states: 772
Possibly predicate: true
Running time cost: 378ms

9.7.2.2 Predicate 2

node1.var4==11

9.7.2.2.1 Simulated system

9.7.2.2.1.1 *General result (possibly predicate)*

Real time period of the log file: 0.000000- 0.370622
Total local states of node1: 8
Total local states of node2: 8
Total local states of node3: 12
Total local states of node4: 12
Total number global states: 530
Possibly predicate: true
Running time cost: 302ms

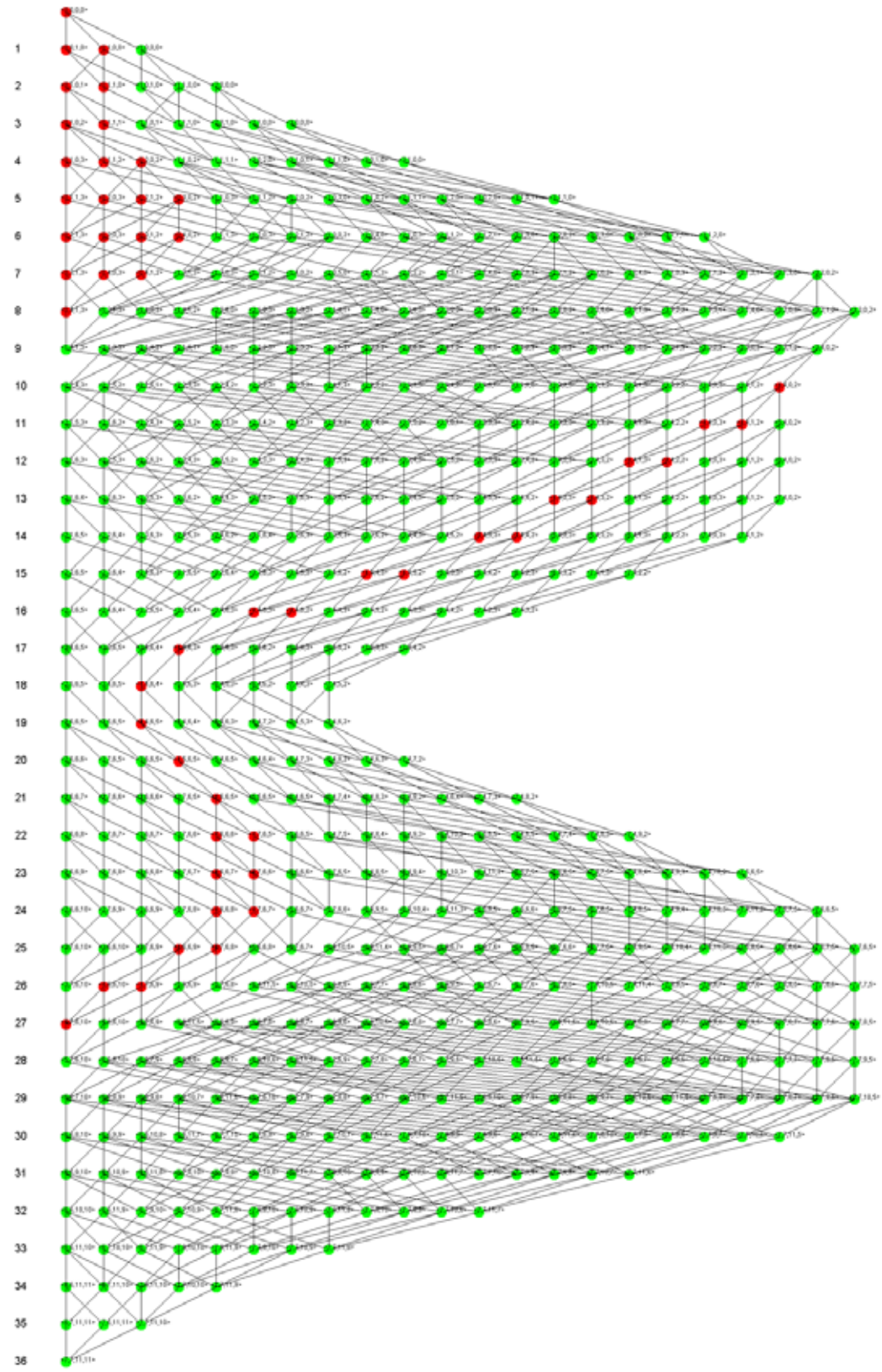


Figure 9-44 graphic result

9.7.2.2.1.2 *General result*

Real time period of the log file: 1.071626- 1.571062
Total local states of node1: 8
Total local states of node2: 10
Total local states of node3: 9
Total local states of node4: 6
Total number global states: 800
Possibly predicate: true
Running time cost: 443ms

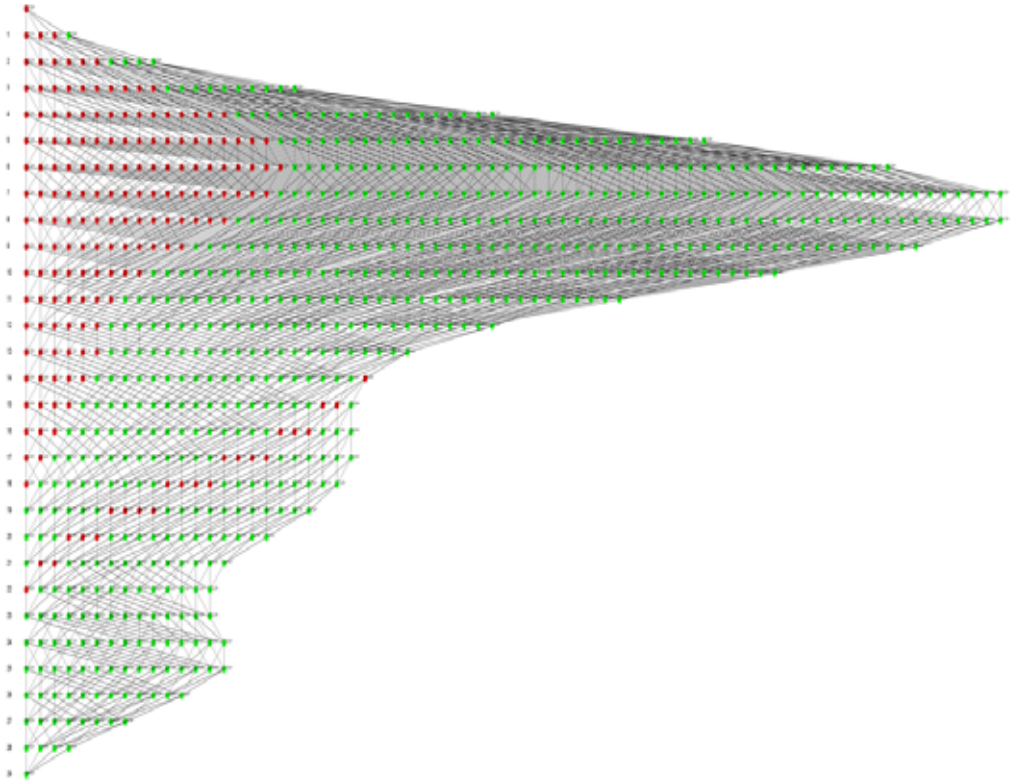


Figure 9-45 graphic result

9.7.2.2.1.3 *General result (definitely predicate)*

Real time period of the log file: 0.000000- 0.370622
Total local states of node1: 8
Total local states of node2: 8
Total local states of node3: 12
Total local states of node4: 12
Total number global states: 530
Definitely predicate: true
Running time cost: 334ms

9.7.2.2.1.4 *General result*

Real time period of the log file: 1.071626- 1.571062
Total local states of node1: 8
Total local states of node2: 10
Total local states of node3: 9
Total local states of node4: 6
Total number global states: 800
Definitely predicate: true
Running time cost: 390ms

9.7.3 Result analysis

This test case includes four nodes. The CANoe log file used in this test case is large. It stores about 12 minutes CANoe data. The total local states of each node are 196, 245, 249, and 200. So the total number of consistent global state evaluations is the product of these numbers. The result of the product is 2,391,396,000. So the prototype should be able to evaluate part of the log file.

This test case tests the data randomly selected from the CANoe log. These data are different periods of the execution data in the CANoe log. They are tested by different predicates and different types of predicate.

This test case finds that in the same predicate and different period execution data, the results of possibly predicate evaluation are the same, so are the results of definitely predicate evaluation. Comparing the graphic lattices of these executions, they have very similar shape. All of them have two heaved curves. The bottom heaved curve in Figure 9-44 is higher than the bottom heaved curve in Figure 9-45. It is because of the different execution periods of each test case; however it should not affect the predicate evaluation result.

9.8 Test case 6

This test case consists of four state machines. They do not communicate with each other, so it does not matter if the test case is tested under the simulated bus or real bus. The total number of the global states should be the product of the total local state of each node.

9.8.1 Model explanation

9.8.1.1 State machine diagram

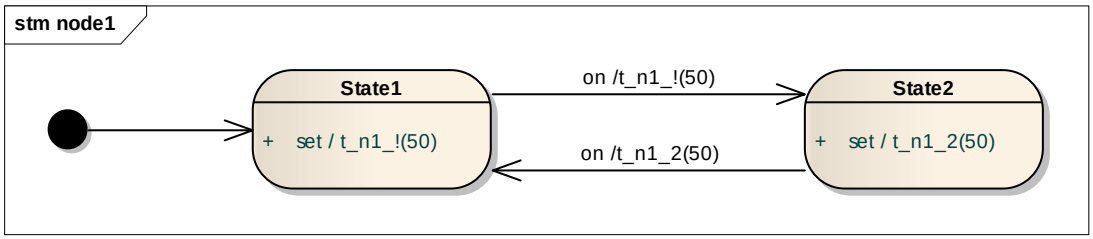


Figure 9-46 test case 6 state machine 1

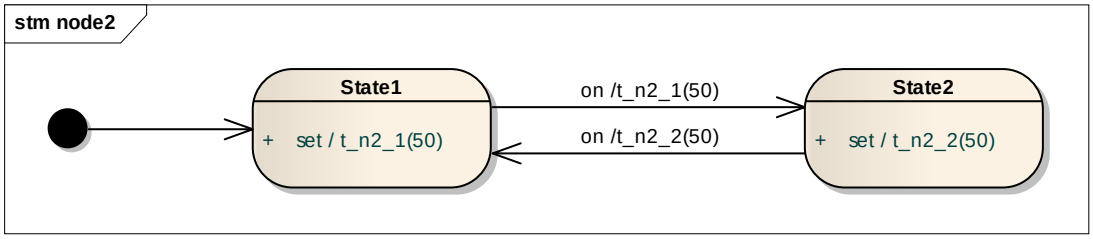


Figure 9-47 test case 6 state machine 2

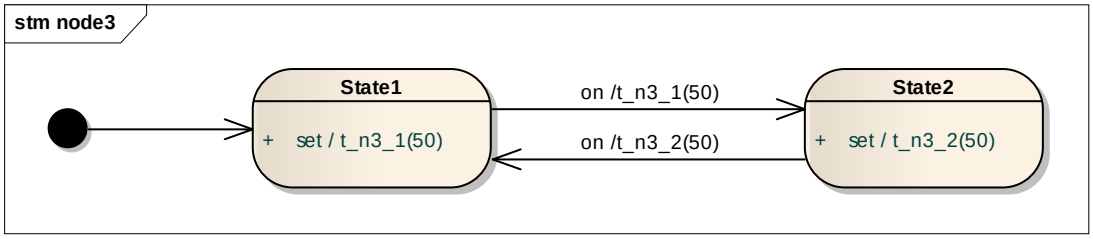


Figure 9-48 test case 6 state machine 3

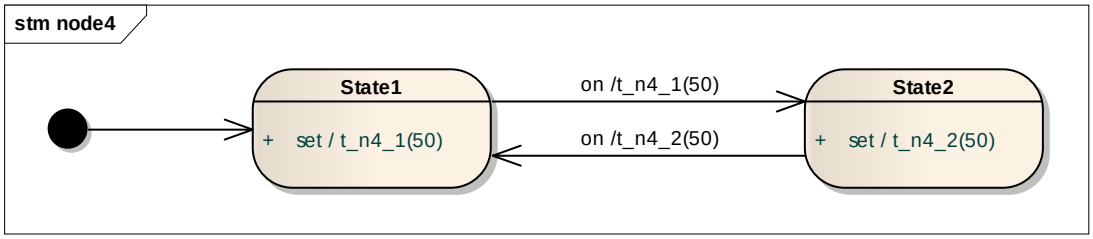


Figure 9-49 test case 6 state machine 4

9.8.1.2 Communication matrix

9.8.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	0	0	0
2	21	22	23	24	25	26	27	28	29	0

Table 9-38 test case 6 each node local states

9.8.2 Test different inputs

9.8.2.1 Predicate 1

9.8.2.1.1 Simulated system test result

node3.var1 < node2.var2 || node4.var2 < node1.var7 || node4.var7 <= 0

&& node2.var4==node1.var6

9.8.2.1.1.1 *General result (possibly predicate)*

Real time period of the log file: 0-0.150926
Total local states of *node1*: 4
Total local states of *node2*: 4
Total local states of node3: 4
Total local states of node4: 4
Total number global states: 256
Possibly predicate: false
Running time cost: 368ms

9.8.2.1.1.2 *General result (possibly predicate)*

Real time period of the log file: 0-0.400956
Total local states of *node1*: 9
Total local states of *node2*: 9
Total local states of node3: 9
Total local states of node4: 9
Total number global states: 6561
Possibly predicate: false
Running time cost: 7157ms

9.8.2.1.1.3 *General result (possibly predicate)*

Real time period of the log file: 0-0.750926
Total local states of *node1*: 16
Total local states of *node2*: 16
Total local states of node3: 16
Total local states of node4: 16
Total number global states: 65536
Possibly predicate: false
Running time cost: 273754ms

9.8.2.1.1.4 *General result (possibly predicate)*

Real time period of the log file: 0-0.850926
Total local states of *node1*: 18
Total local states of *node2*: 18
Total local states of node3: 18
Total local states of node4: 18
Total number global states: 104976
Possibly predicate: false
Running time cost: 627407ms

9.8.3 Result analysis

This test case includes three nodes. They do not communicate to each other, so all global states evaluated are consistent. The total global state should be the product of total local states of each node. This is proven by the results of this test case. The storage used by the lattice is extremely high. The Java virtual machine crashed with such big memory usage and a memory out of boundary exception was thrown.

9.9 Test case 7

This test case consists of six state machines. It tests the ability of the prototype to evaluate large numbers of state machines.

9.9.1 Model explanation

9.9.1.1 State machine diagram

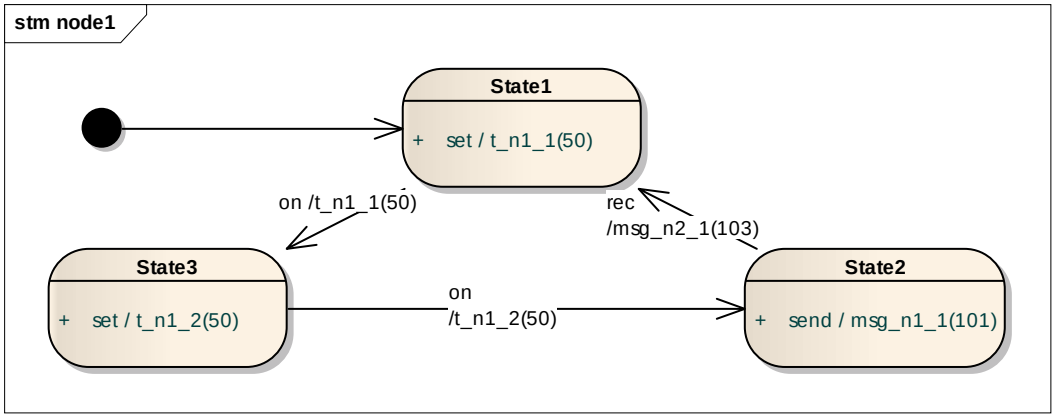


Figure 9-50 test case 7 state machine 1

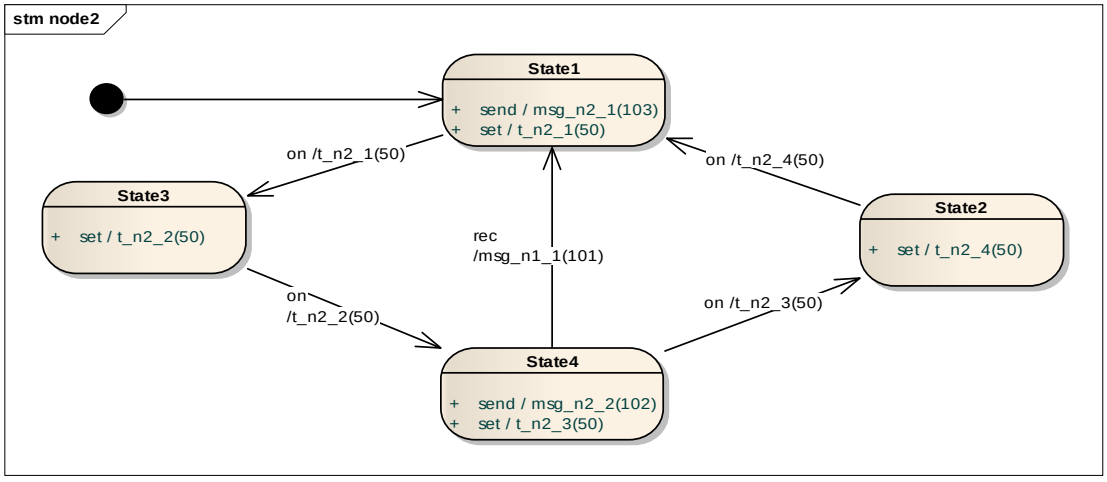


Figure 9-51 test case 7 state machine 2

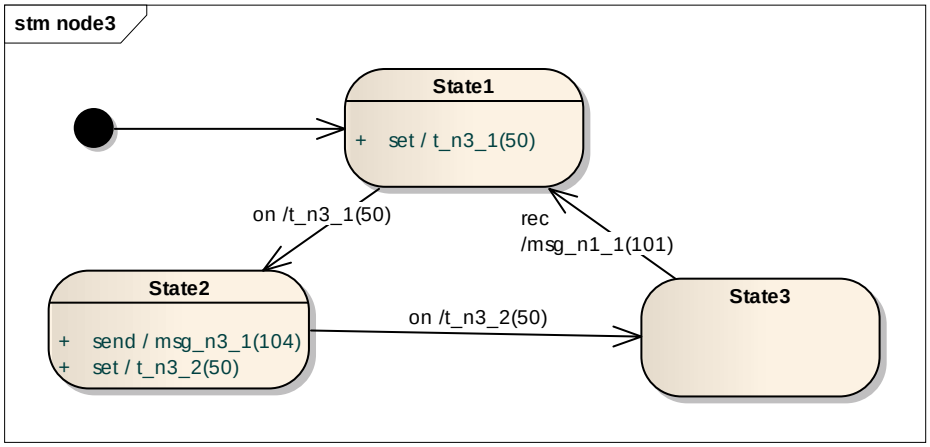


Figure 9-52 test case 7 state machine 3

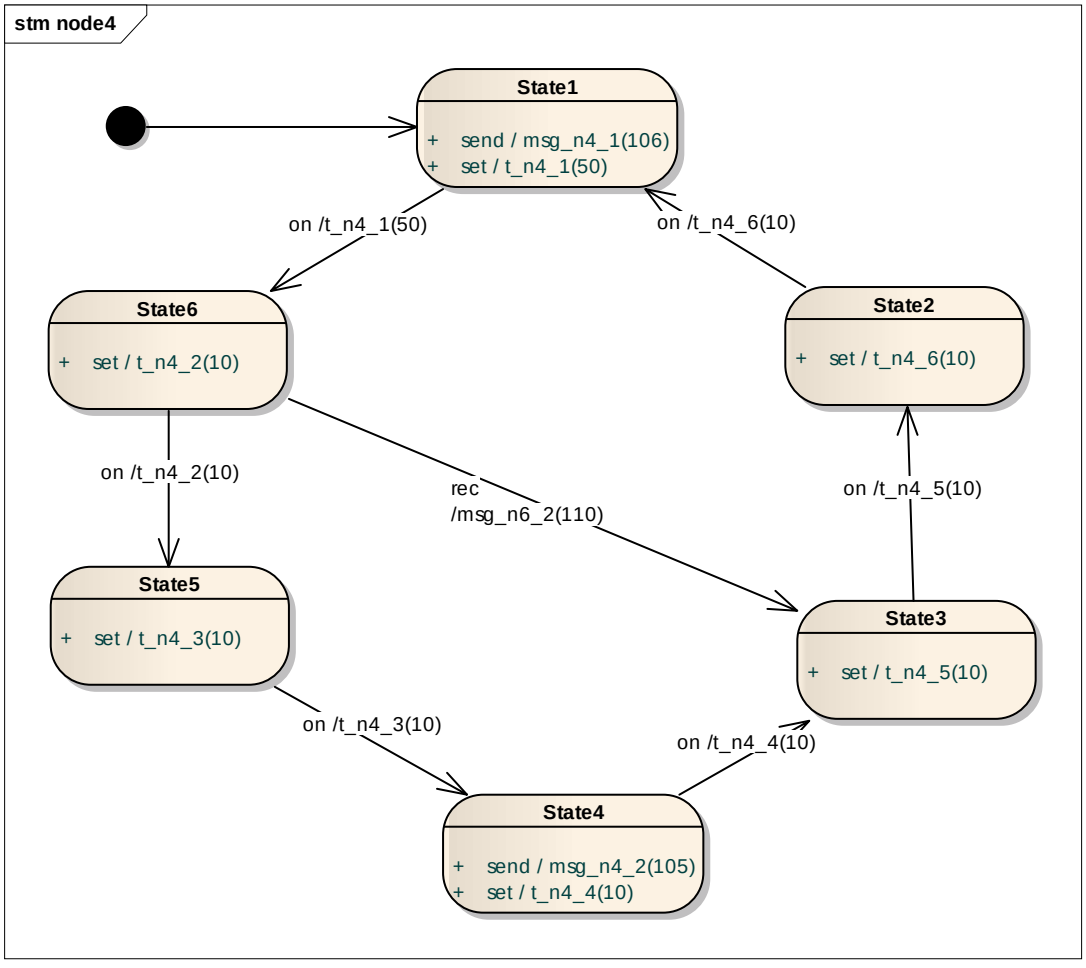


Figure 9-53 test case 7 state machine 4

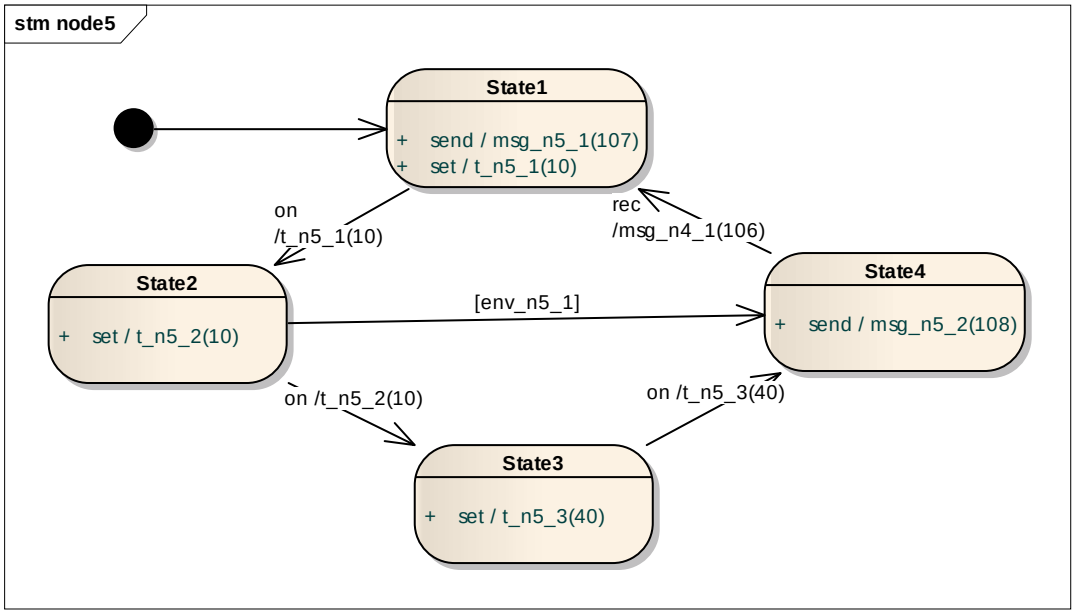


Figure 9-54 test case 7 state machine 5

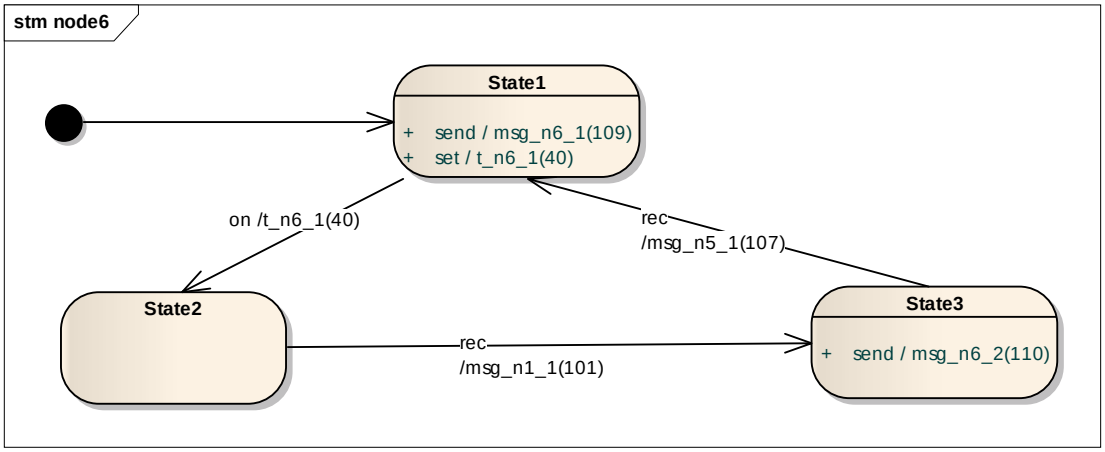


Figure 9-55 test case 7 state machine 6

9.9.1.2 Communication matrix

MessageID	SendNodeNum	nodeNum
110	6	4
101	1	2
101	1	3
101	1	6
102	2	
103	2	1
104	3	
105	4	
106	4	5
107	5	6
108	5	
109	6	

Table 9-39 test case 7 communication matrix

9.9.1.3 Local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	18	0	0
2	21	22	23	24	25	26	27	28	29	0
3	31	32	33	34	35	36	37	38	39	0

Table 9-40 test case 7 node 1 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	18	0	0
2	21	22	23	24	25	26	27	28	29	0
3	31	32	33	34	35	36	37	38	39	0
4	41	42	43	44	45	46	47	48	49	0

Table 9-41 test case 7 *node2* local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	11	11	11	11	11	11	11	0	0
2	21	21	21	21	21	21	21	21	0	0
3	31	31	31	31	31	31	31	31	31	0

Table 9-42 test case 7 node 3 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	18	0	0
2	21	22	23	24	25	26	27	28	29	0
3	31	32	33	34	35	36	37	38	39	0
4	41	42	43	44	45	46	47	48	49	0
5	51	52	53	54	55	56	57	58	59	0
6	61	62	63	64	65	66	67	68	69	0

Table 9-43 test case 7 node 4 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	18	0	0
2	21	22	23	24	25	26	27	28	29	0
3	31	32	33	34	35	36	37	38	39	0
4	41	42	43	44	45	46	47	48	49	0

Table 9-44 test case 7 node 5 local states

stateNum	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10
1	11	12	13	14	15	16	17	18	0	0
2	21	22	23	24	25	26	27	28	29	0
3	31	32	33	34	35	36	37	38	39	0

Table 9-45 test case 7 node 6 local states

9.9.2 Test different inputs

node4.var2<=11 || *node2*.var4==*node1*.var6

9.9.2.1 Predicate 1

9.9.2.1.1 Simulated system test result

9.9.2.1.1.1 *General result (possibly predicate)*

Real time period of the log file: 0-0.070364
Total local states of *node1*: 2
Total local states of *node2*: 3
Total local states of node3: 3
Total local states of node4: 6
Total local states of node5: 6
Total local states of node6: 3
Total number global states: 1620
Possibly predicate: false
Running time cost: 1226ms

9.9.2.1.1.2 *General result (possibly predicate)*

Real time period of the log file: 0-0.101776
Total local states of *node1*: 4
Total local states of *node2*: 5
Total local states of node3: 5
Total local states of node4: 9
Total local states of node5: 6
Total local states of node6: 4
Total number global states: 10800
Possibly predicate: false
Running time cost: 19570ms

9.9.2.1.1.3 *General result (possibly predicate)*

Real time period of the log file: 3.960232-4.370364
Total local states of *node1*: 16
Total local states of *node2*: 20
Total local states of node3: 16
Total local states of node4: 34
Total local states of node5: 26
Total local states of node6: 20
Total number global states: 221017
Possibly predicate: false
Running time cost: 799612ms

9.9.2.1.2 Real system test result

9.9.2.1.2.1 *General result (possibly predicate)*

Real time period of the log file: 43.459959-43.578460
Total local states of *node1*: 4
Total local states of *node2*: 5
Total local states of node3: 4
Total local states of node4: 10
Total local states of node5: 7
Total local states of node6: 2
Total number global states: 2352
Possibly predicate: false
Running time cost: 1767ms

9.9.2.1.2.2 *General result (possibly predicate)*

Real time period of the log file: 43.459959-43.848486
Total local states of *node1*: 13
Total local states of *node2*: 16
Total local states of node3: 14
Total local states of node4: 31
Total local states of node5: 26
Total local states of node6: 15
Total number global states: 114014
Possibly predicate: false
Running time cost: 292335ms

9.9.3 Result analysis

This test case consists of six nodes. The possible number of combination of consistent global state is the product of total number of these six nodes local state. Even if node has a small number of local states, the product is going to be large. Also it is possible to store lots of consistent global states. As in section 9.9.2.1.1.1 the total number of local state of each node is 2, 3, 3, 6, 6, and 3. The total number of global state is 1620. In section 9.9.2.1.1.3, the total number of local states of each node is 16, 20, 16, 34, 26, and 20. The total number of consistent global states is 221017. It takes about 799612 million seconds (13.33 minutes). The

results show the storage and time consumption is very large for a large number of nodes.

9.10 Prototype Performance Analysis

9.10.1 Memory consumption

The prototype program is coded in Java language, which runs on the JVM (Java Virtual Machine). During the test, the JVM crashed few times by the exception of memory out of boundary. Therefore, it is necessary to analysis how much memory that is taken by the lattice. Table 9-46 (Borland Software Corporation 2005, p8-9) shows the Java memory consumption of each primitive data type.

Type	Size (byte)
double	8
int	4
long	8
float	4
short	2
byte	1
char	2

Table 9-46 Java primitive data type memory consumption

Because a global state is constructed from the local states, to know the memory size of a local state is essential. By comparing Table 9-46 and the attributes of the *LocalState* class, a size of a local state is at least 114 bytes for a two nodes system (the size of *vectorTime* depends on how many nodes on the system. The two nodes system is the smallest distributed system). Let *t_node* denotes the total number of nodes. The size of a local state can be calculated by using:

Memory_size_of _a_ local_state =

(98+ (*t_node* × 8))

If without the *vectorTime* the size of a local state is 98 bytes. Except the local states and global vector time, a *GlobalState* class consists of other attributes, their size is 8.25 byte (a Boolean value is one bit). It equals to 66 bits. The size for a global state will be calculated by using:

Memory_size_of _a_ global_state =

$$((98 + (t\_node \times 8)) \times t\_node) + 8.25 + (t\_node \times 8)$$

A global state of a 10-node system will consume 1868.25 bytes. If 100 consistent global states are found, the storage will consume 186825 bytes memory.

Because most memory is used by storing the consistent global states, it is necessary to find what factors produce the consistent global states. Table 9-47 shows all memory consumptions for all test cases.

9.10.2 Factors affecting the quantity of consistent global states

Because a global state is constructed by the local states and all the consistent global states are evaluated from these global states, the number of local states is the main factor influences the quantity of consistent global states.

Depending on the results of the test cases, the quantity of consistent global states affected by the following factors:

- The number of the nodes are an obvious factor that influences the quantity of the consistent global states. The bigger system (more nodes) the more consistent global states are generated. The bigger system, the more complex of the execution condition of the system will be and the more local states will be generated; thereby the more candidates global states will be constructed by these local state. From the large population of

global state to evaluate the consistent global state, there is a big chance to achieve the large number of consistent global states.

- The communications between the nodes are another factor. The more communications the less consistent global states will be. Because sometimes, a node moving state needs to receive a message, if the message is not transmitted, the local state of the node will not be changed, thereby it reduces the number of the local states. It reduces the candidates for the consistent global state evaluation.
- The number of consistent global states also can depend on the interval of the system running time. If the transition of local states is more depending on the timer expiring, then the longer the system running, the more local states will generated. If the transition more depends on the event happen, then the time will not affect the quantity of local states, thereby it will not affect the quantity of the consistent global state.

Figure 9-56 is generated from the results of the test cases (Table 9-47); the 3-dimension plot shows how the number of nodes and number of communications influence the number of consistent global state. Test case 5 and test case 6 has the same number of nodes. Comparing the number of consistent global states between them in Figure 9-57, the test case 6 is higher than the test case 5; because test case 5 has more communications than test case 6 as shown in Figure 9-58.

Test case 5 and test case 7 has the same number of communications. Comparing the number of consistent global state between them in Figure 9-59, the test case 7 is higer than the test case 5; because test case 5 has less nodes than test case 7 as shown in Figure 9-58.

Most test cases are based on the time triggering system (except test case 3), thereby for the same test case in Figure 9-56, the longer system running the more consistent global states are generated.

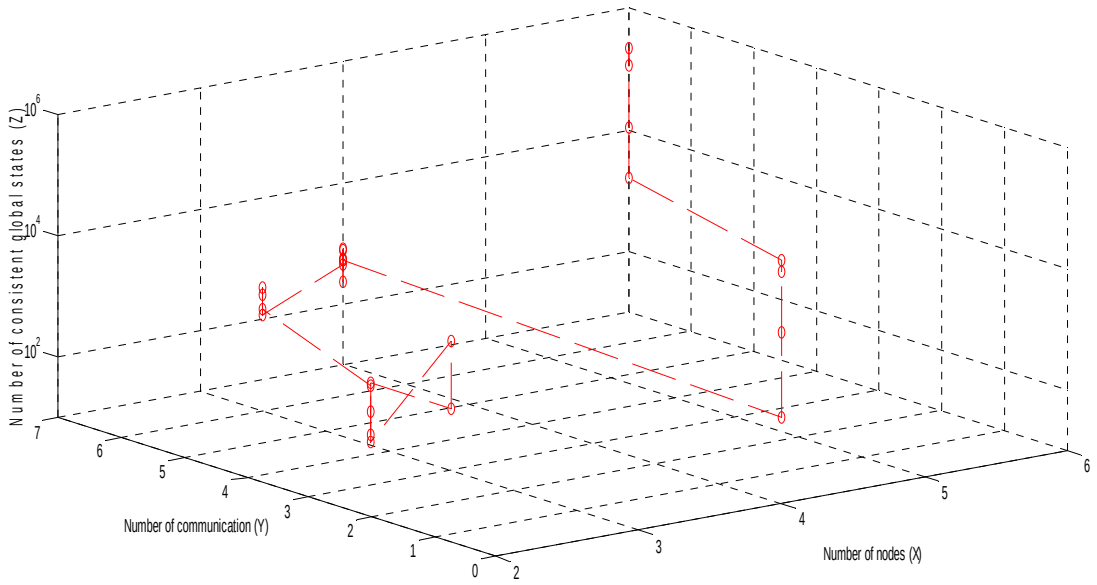


Figure 9-56 number of CGSs vs. number of communication & number of nodes

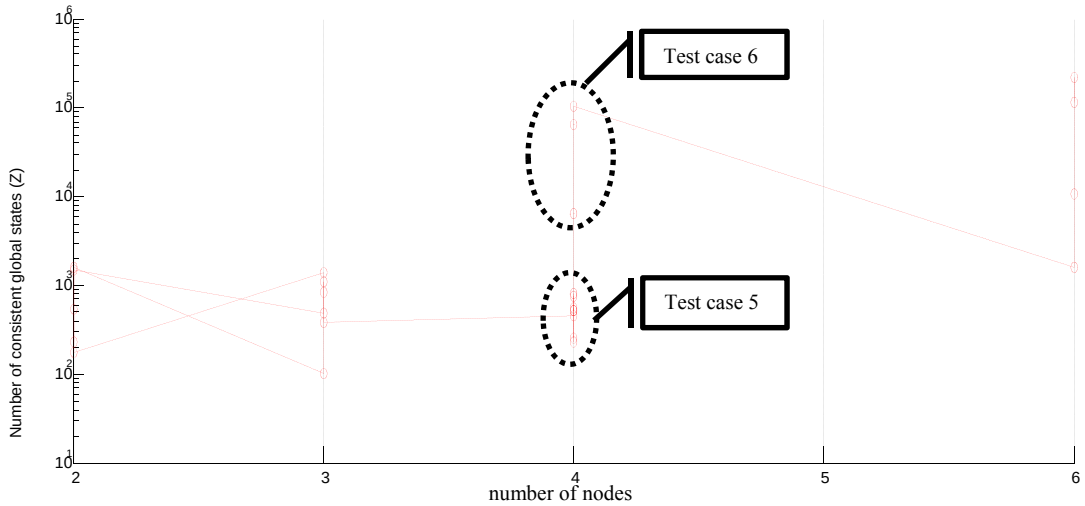


Figure 9-57 number of nodes vs. number of CGSs (X-Z view of Figure 9-56)

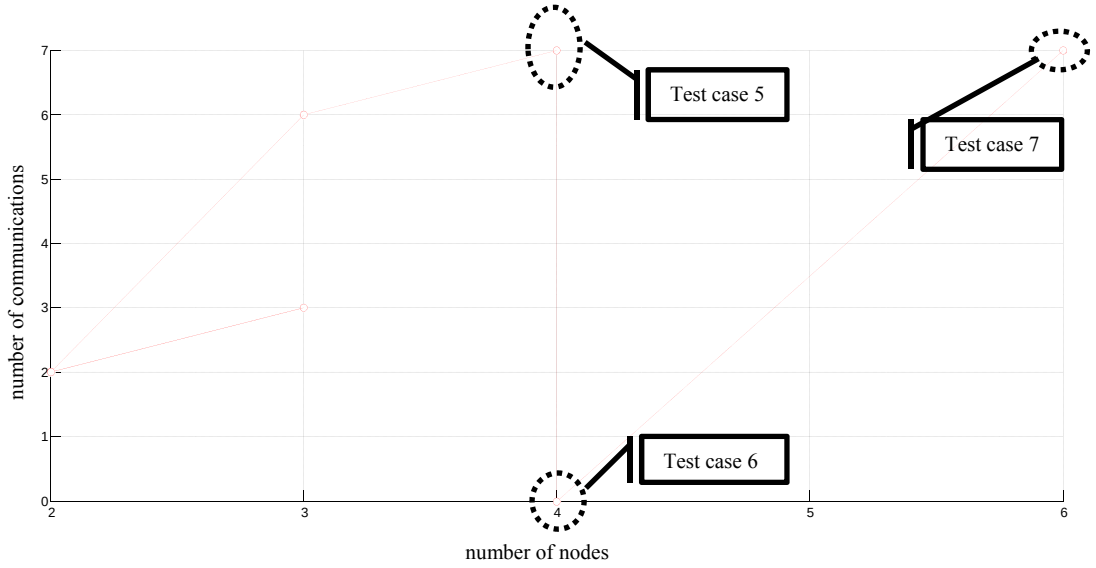


Figure 9-58 number of communications vs. number of nodes (X-Y view of Figure 9-56)

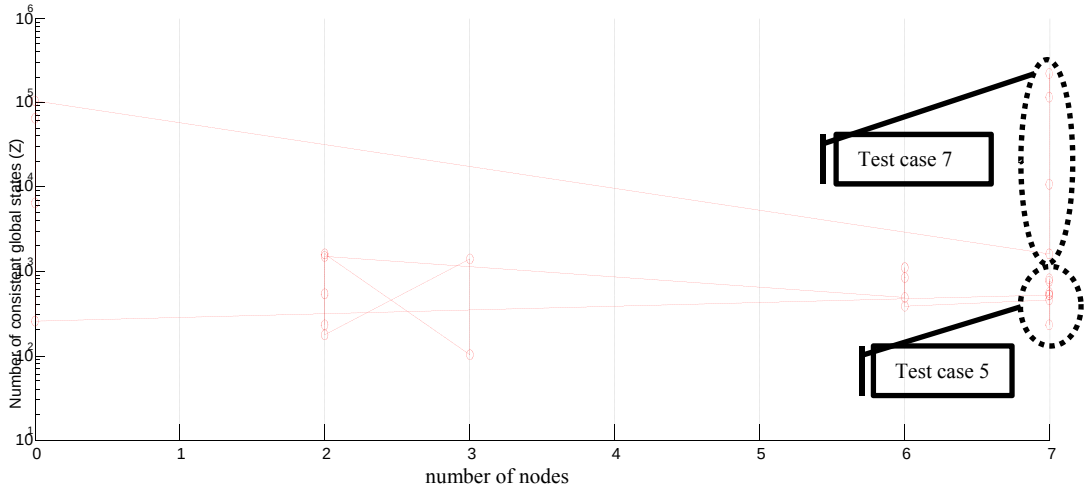


Figure 9-59 number of communications vs. number of CGSs (Y-Z view of Figure 9-56)

Number of Communications	Number of nodes	Consistent Global States	Memory consumption (byte)
2	2	234	59026.5
2	2	542	136719.5
2	2	1626	410158.5
3	3	103	41019.75
3	3	1405	559541.3
2	2	176	44396
2	2	1493	376609.3
6	3	492	195939
6	3	833	331742.3
6	3	1097	436880.3
6	3	381	151733.3
7	4	454	254353.5
7	4	566	317101.5
7	4	234	131098.5
7	4	772	432513
7	4	530	296932.5
7	4	800	448200
7	4	530	296932.5
0	4	256	143424
0	4	6561	3675800
0	4	65536	36716544
0	4	104976	58812804
7	6	1620	1510245
7	6	10800	10068300
7	6	221017	2.06E+08
7	6	114014	1.06E+08

Table 9-47 number of nodes, number of communications, number of CGS and memory consuming from all test cases results

9.11 Conclusion

This chapter describes seven test cases to test the prototype software. The first four test cases verify and validate the prototype. The last three test cases test the performance of the prototype. Analysing each test case result, for the verification of the functions of the prototype is successful. The validation of the prototype is also successful. The performance of the prototype depends on the communications between nodes, the total number of nodes, and the total number of each node's local states.

References

Borland Software Corporation. Getting Started with Java? 2005.

Vector. Programming With CAPL. 12-14-2004. Vector CANtech, Inc.

Section Five: Research Summary

Chapter 10 Research conclusion

10.1 Introduction

The aim of this research is to investigate a method to test distributed automotive system. To achieve this aim, the system global states are constructed and evaluated by the specified predicates.

10.2 Research summary

There are some literatures investigated for testing the distributed system. The logical time is used to order the distributed events. The snapshot algorithm capture a global state of an execution, but it cannot continuously capture the global states. However for distributed system testing the continuous history of the execution is important, so it leads to the GPD algorithm.

By comparing and contrasting different GPD algorithms, the centralized relational predicate is chosen to apply on the automotive distributed system. A prototype program was developed to evaluate the global predicates. During the prototype development, seven test cases were used to verify and validate the prototype. The prototype was successfully verified and validated in these test cases.

10.3 Answer Research Questions

10.3.1 How can events occurring on separate ECUs be chronologically ordered?

The events occurring on the different ECUs can be ordered by the vector clock as in the prototype program. The prototype assigns the vector clock to each local

state of each ECU; the test cases verified the local vector clock assignment working fine.

10.3.2 How can a snapshot of the global application state and application execution traces be constructed based on test case execution cycles?

Depending on vector clock of each local state, the consistent global states can be constructed. Each consistent global state also is assigned the vector clock, the vector clock of the consistent global states assignment is verified by the test cases.

10.3.3 How to deal with the large number of global states of execution?

The global states include consistent global states and non-consistent global states. More global states consume more CPU power to evaluate if these global states are consistent. These consistent global states need to be stored in a data structure such as the execution lattice. The data structure could consume lots of memory. All problems stem from the large number of global states.

The total number of global states equals the product of total number of each node local states. The number of nodes can't be reduced. The entire system should be evaluated. The only way is to reduce the local states of each node. The local states are constructed by the CAN messages from the CANoe log file. The longer time the system run, the more data are collected. So if the running time of the testing system is reduced, it will reduce the total number of local states. If the system is run for a short time, it may work well, but it won't be certain if it works for a long time running. So the prototype should have the ability to evaluate a period time of the system running. It should allow the user to choose the part of the CANoe log

file to be evaluated. So the best way to deal with the large number of global states is to evaluate the local states over a series of shorter intervals.

10.4 Area for further research

Due to the time constraint for the research, the prototype is not a sophisticated product. Lots of optimizations and extensions can be developed in the further. They are:

- In the prototype program, the algorithm used for evaluating the consistent global state is checking every possible global state. In the situation, where one or more global states caused by an inconsistent parent global state; there is no need to evaluate its child global states. The time to evaluate the consistent global states can be shortened. If an algorithm offers the ability to cleverly skip all child global states of a non-consistent global state, it will be very useful for large distributed automotive systems.
- For the execution lattice presentation, the prototype presents the lattice on the JAVA frame; the user can browse the global state on the lattice by double clicking. The lattice is also can be saved as PNG format, but the PNG format cannot interact with the user, it only a picture. If too many global states are on the lattice, the lattice cannot be presented on the frame or PNG image. So finding a way to present large lattices in a different format that can offer a good user interface would let the user easily browse the global state information.
- For the performance analysis of the prototype program, some mathematic al solution may be generated by the results data of massive test cases.

Because the number of consistent global states is affected by the number of nodes, the number of communications, and maybe the system running period (the local state transition depends on the timer); it is may be possible to use the massive test results' data to figure out the coefficient of these factors and generate a formula.

Appendix A

Bibliography

A.Al-Ashaab, S.Howell, A.Gorka, K.Usowicz, & P.Hernando Anta Set-Based Concurrent Engineering Model for Automotive Electronic/Software Systems Development, In CIRP Design Conference 2009, p. 464.

Ajay D.Kshemkalyani. A Fine-Grained Modality Classification for Global Predicates. 14, 807-816. 2003. IEEE Transactions.

Alexander I.Tomlinson & Vijay K.Garg. Detecting relational global predicates in distributed system. 28[12]. 1993. ACM New York, NY, USA.

AUTOSAR GbR. requirements of RTE. 7-12-2006a.

AUTOSAR GbR. software component template. 6-26-2006b.

AUTOSAR GbR. Specification of PDU Router. 6-26-2006c.

AUTOSAR GbR. Layered Software Architecture. 2-14-2008a.

AUTOSAR GbR. Specification of Communication. 2-13-2008b.

AUTOSAR GbR. Specification of Operating System. 6-23-2008c.

AUTOSAR GbR. Specification of RTE. 9-22-2010.

Borland Software Corporation. Getting Started with Java? 2005.

BOSCH. CAN Specification. 1991. Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart.

Bowen, J. 2003. Formal Specification and Documentation using Z: A Case Study Approach Thomson Publishing.

Brendan Jackman. Basic Concepts. An overview of the distinguishing features of the CAN network. 2004a. Waterford Institute of Technology, Ireland.

Brendan Jackman. CAN Frame Formats. 2004b. Waterford Institute of Technology, Ireland.

Broekman, B. & Notenboom, E. 2003. Testing Embedded Software.

Colin Fidge. Logical time in distributed computing systems. 28-33. 1991. IEEE Computer.

Darren Buttle. What is an RTE. Introduction to AUTOSAR for RTE users. 12-5-2005.

Dasso, A. & Funes, A. 2006. Verification, Validation and Testing in Software Engineering Idea Group Publishing.

dSPACE GmbH. HIL for a Three-Wheeler Scooter. 2007. dSPACE .

dSPACE GmbH. dSPACE Catalog 2009. 2009. dSPACE.

dSPACE GmbH. Making Power Windows safe. 2010. dSPACE GmbH, Paderborn, Germany, dSPACE Magazine.

dSPACE, G. dSPACE Catalog 2009. 2009. dSPACE,GmbH.

Everett, G.D. & McLeod, R. 2007. Software Testing Testing Across the Entire Software Development Life Cycle Wiley-IEEE Computer Society Press.

Friedemann Mattern. Algorithms for distributed termination detection. 2, 161-175. Distributed Computing.

Friedemann Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. 18, 423-434. 1993. Journal of Parallel and Distributed Computing.

Friedemann Mattern. Virtual time and global states of distributed systems. 215-226. 1998. Proceedings of the Parallel and Distributed Algorithms Conference.

G.T.J.Wuu & A.J.Bernstein. Efficient solutions to the replicated log and dictionary problems. 233-242. 1984. Proceedings of 3rd ACM Symposium on PODC.

Gabriel Leen & Donal Heffernan. Expanding Automotive Electronic Systems. 88-93. 2002. IEEE.

Gary S.Ho & C.V.Ramamoorthy. Protocols for Deadlock Detection in Distributed Database Systems. 8. 1982. IEEE Transactions On Software Engineering.

H.Kleinknecht. CAN Calibration Protocol Version 2.1. 2-18-1999.

Joseph Lemieux. Programming in the OSEK/VDX Environment. Berney Williams, Robert Ward, Rita Sooby, and Michelle O'Neal. 2001. CMP Books.

K.M.Chandy & L.Lamport. Distributed Snapshots: Determining Global: States of Distributed Systems. 3. 1985. ACM Transactions on Computer Systems.

Keith Marzullo & Gil Neiger. Detection of Global State Predicates. [LNCS 579], 254-272. 1991. Proceedings of the 5th Workshop on Distributed Algorithms.

Kenneth P.Birman. Building Secure and Reliable Network Applications. 1995. Department of Computer Science Cornell University Ithaca, New York 14853.

Leslie Lamport. Time clocks and the ordering of events in a distributed system. 558-564. 1978. Communications of the ACM.

LiveDevices Ltd. RTA-RTE User Guide. 2004.

M.J.Fischer & A.Michael. Sacrifying serializability to attain hight availability of data in an unreliable network. 70-75. 1982. Proceedings of the ACM Symposium on Principles of Database Systems.

Madalene Spezialetti & Phil Kearns. Efficient Distributed Snapshots. 382-388. 1986. Proceedings of the 6th International Conference on Distributed Computing Systems.

Maria, A. Introduction To Modeling And Simulation. 1997. ACM.

McGregor, J.D. & Sykes, D.A. 2001. A practical guide to testing object-oriented software Addison-Wesley.

Michael Schneider, Johnny Martin, & W.T.Tsai. An Experimental Study of Fault Detection In User Requirements Documents. 1, 188-204. 1992. ACM Transactions on Software Engineering and Methodology.

Nathan Funk. Jep Java Math Expression Parser. 2-8-2011.

National ITS Architecture Team 2007, System Engineering for Intelligent Transportation Systems.

Neeraj Mittal & Vijay K.Garg. On Detecting Global Predicates in Distributed Computations. 3-10. 2001. International Conference on Distributed Computing Systems.

Nicola Santoro. Design And Analysis Of Distributed Algorithms. 2007. John Wiley & Sons, Inc., Hoboken, New Jersey.

Nicolas Navet & Françoise Simonot-Lion. Automotive Embedded Systems Handbook. 2009. Taylor & Francis Group, LLC.

OSEK. OSEK/VDX Operating System Specification 2.2.3. 2-17-2005.

OSEK/VDX. OSEK/VDX Communication. 7-20-2004.

Ozalp Babaoglu & KeithMarzullo . Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. 1993. Italy, Laboratory for computer science university of bologna.

Patton, R. 2005. Software Testing Sams Publishing.

Pressman, R.S. 2001. Software Engineering, 5th ed. Thomas Casson.

Punit Chandra & Ajay D.Kshemkalyani. Distributed algorithm to detect strong conjunctive predicates. 87, 243-249. 2003. Information Processing Letters.

Punit Chandra & Ajay D.Kshemkalyani. Causality-Based Predicate Detection across Space and Time. 54, 1438-1453. 2005. IEEE Transactions on Computers.

Rahul Garg, Vijay K.Garg, & Yogish Sabharwal. Efficient Algorithms for Global Snapshots in Large Distributed Systems. 1994. IEEE Transactions on Software Engineering.

Rainer Zaiser. CCP. A CAN Protocol for Calibration and Measurement Data Acquisition. 2011. Vector Informatik GmbH Friolzheimer Strasse 6 70499 Stuttgart,Germany.

Ranal, M. & Singhal, M. Logical Time: Capturing Causality in Distributed Systems. 1996. IEEE Computer.

Richard Zurawski. networked embedded systems. 2009. Taylor & Francis Group, LLC.

Robert Cooper & Keith Marzullo. Consistent Detection of Global Predicates. 163-173. 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

Robert Warschofsky. AUTOSAR Software Architecture. 2011. Hasso-Plattner-Institute fuer Softwaresystemtechnik.

Roychoudhury, A. 2009. Embedded Systems and Software Validation Morgan Kaufmann.

Schaeuffele, J. & Zurawka, T. 2005. Automotive Software Engineering Principles Processes Methods and Tools SAE International.

Simon Fuerst & BMW AUTOSAR An open standardized software architecture for the automotive industry, In 1st AUTOSAR Open Conference & 8th AUTOSAR Premium Member Conference.

Stefan Bunzel. Overview on AUTOSAR Cooperation. 5-13-2010. Tokyo, Japan, 2nd AUTOSAR Open Conference.

Sukumar Ghosh. Distributed Systems An Algorithmic Approach. 2007. Taylor & Francis Group, LLC Chapman & Hall/CRC.

T.H.Lai & T.H.Yang. On distributed snapshots. 25, 153-158. 1987. Information Processing
Letters.

Tian, J. 2005. Software Quality Engineering Testing, Quality Assurance, and Quantifiable Improvement John Wiley & Sons, Inc., Hoboken, New Jersey.

Vector. Programming With CAPL. 12-14-2004. Vector CANtech, Inc.

Vector CANtech, I. Programming With CAPL. 12-14-2004.

Vector Informatik GmbH. Product Catalog. Development of Distributed Systems ECU Testing. 2010a. Vector Informatik GmbH.

Vector Informatik GmbH. User Manual CANoe Version 7.5. 2010b.

Vijay K.Garg & Brian Waldecker. Detection of weak unstable predicates in distributed programs. 5, 299-307. 1994. IEEE Transactions on Parallel and Distributed Systems.

Vijay K.Garg & Brian Waldecker. Detection of Strong Unstable Predicates in Distributed Programs. 7, 1323-1333. 1996. IEEE Transactions on Parallel and Distributed Systems.