Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Hierarchical runtime verification for critical cyber-physical systems

Scientific Students' Associations Report

Authors:

László Balogh
Flórián Deé
Bálint Hegyi

Supervisors:

dr. István Ráth
dr. Dániel Varró
András Vörös

2015.

Contents

Összefoglalás Ipari becslések szerint 2020-ra 50 milliárdra nő a különféle okoseszközök száma, amelyek egymással és velünk kommunikálva komplex rendszert alkotnak a világhálón. A szinte korlátlan kapacitású számítási felhőbe azonban az egyszerű szenzorok és mobiltelefonok mellett azok a kritikus beágyazott rendszerek – autók, repülőgépek, gyógyászati berendezések - is bekapcsolódnak, amelyek működésén emberéletek múlnak. A kiberfizikai rendszerek radikálisan új lehetőségeket teremtenek: az egymással kommunikáló autók baleseteket előzhetnek meg, az intelligens épületek energiafogyasztása csökken.

A hagyományos kritikus beágyazott rendszerekben gyakorta alkalmazott módszer a futási idejű ellenőrzés. Ennek célja olyan ellenőrző programok szintézise, melyek segítségével felderíthető egy kritikus komponens hibás, a követelményektől eltérő viselkedése a rendszer működése közben.

Kiberfizikai rendszerekben a rendelkezésre álló számítási felhő adatfeldolgozó kapacitása, illetve a különféle szenzorok és beavatkozók lehetővé teszik, hogy több, egymásra hierarchikusan épülő, különböző megbízhatóságú és felelősségű ellenőrzési kört is megvalósíthassunk. Ennek értelmében a hagyományos, kritikus komponensek nagy megbízhatóságú monitorai lokális felelősségi körben működhetnek. Mindezek fölé (független és globális szenzoradatokra építve) olyan rendszerszintű monitorok is megalkothatók, amelyek ugyan kevésbé megbízhatóak, de a rendszerszintű hibát prediktíven, korábbi fázisban detektálhatják.

A TDK dolgozatban egy ilyen hierarchikus futási idejű ellenőrzést támogató, matematikailag precíz keretrendszert dolgoztunk ki, amely támogatja (1) a kritikus komponensek monitorainak automatikus szintézisét egy magasszintű állapotgép alapú formalizmusból kiindulva, (2) valamint rendszerszintű hierarchikus monitorok létrehozását komplex eseményfeldolgozás segítségével. A dolgozat eredményeit modellvasutak monitorozásának (valós terepasztalon is megvalósított) esettanulmányán keresztül demonstráljuk, amely többszintű ellenőrzés segítségével képes elkerülni a vonatok összeütközését.

Abstract According to industrial estimates, the number of various smart devices - communicating with either us or each other - will raise to 50 billion, forming one of the most complex systems on the world wide web. This network of nearly unlimited computing power will not only consist of simple sensors and mobile phones, but also cars, airplanes, and medical devices on which lives depend upon. Cyber-physical systems open up radically new opportunities: accidents can be avoided by cars communicating with each other, and the energy consumption of smart buildings can be drastically lower, just to name a few.

The traditional critical embedded systems often use runtime verification with the goal of synthesizing monitoring programs to discover faulty components, whose behaviour differ from that of the specification.

The computing and data-processing capabilities of cyber-physical systems, coupled with their sensors and actuators make it possible to create a hierarchical, layered structure of high-reliability monitoring components with various responsibilities. The traditional critical components' high-reliability monitors' responsibilities can be limited to a local scope. This allows the creation of system-level monitors based on independent and global sensory data. These monitors are less reliable, but can predict errors in earlier stages.

This paper describes a hierarchical, mathematically precise, runtime verification framework which supports (1) the critical components' monitors automatic synthetisation from a high-level statechart formalism, (2) as well as the creation of hierarchical, system-level monitors based on complex event-processing. The results are presented as a case study of the monitoring system of a model railway track, where collisions are avoided by using multi-level runtime verification.

Chapter 1

Introduction

1.1 Cyber-physical systems

The design of complex cyber-physical systems is an interdisciplinary process. From an engineering point of view, defining the requirements, designing sufficiently reliable components, dealing with scalability issues, employing testing processes that result in high test coverage, verifying safety critical components, and maintainability issues are all present at the same time. Systems engineering focuses on how to design and manage such systems. [9] [13]

1.2 Safety Critical System

Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. There are many well known examples in application areas such as medical devices, aircraft flight control, weapons, and nuclear systems. Many modem information systems are becoming safety-critical in a general sense because financial loss and even loss of life can result from their failure. Future safety-critical systems will be more common and more powerful. From a software perspective, developing safety critical systems in the numbers required and with adequate dependability is going to require significant advances in areas such as specification, architecture, verification, and process. The very visible problems that have arisen in the area of information-system security suggests that security is a major challenge also. [8]

1.3 Verification

As modern society is becoming more and more dependent on cyber-physical systems, the need for faultlessly working hardware and software increases. Validation and verification methodologies have been present in the development processes of such systems for a long time [15], but faster and more reliable approaches are needed. Validation checks that the requirements specified for the software meet the needs of the user – as such, validation usually can be aided, but can't entirely be done by software. The point of verification is to check whether the specified requirements are met. Methods for verification can be divided into two groups: design time verification and runtime verification.

1.4 Design time verification

In the traditional software development methods, the verification process is in the design time. This means after the composition of system elements, we verify it's compliance with the specification. One example for this is the formal verification, where we want to proof the match in the behavior of the finished system with the specifications written in mathematically proofed formalism.

1.5 Runtime verification

Runtime verification is based on the inspection of the running software. We watch the behavior of the system, and conforming it to the specification. This has an advantage of detecting non specified behavior after deployment. Runtime and design time verification aren't exclusive, instead of they can support a more robust system verification process. The motive of using runtime verification is the difficulty of designing highly safety systems. These systems are getting bigger in complexity, and traditional methods cannot verify it, or the resource need of a verification of this scale cannot be realized.

1.6 V-Model

A concept of operations is one of the initial stages in a system life cycle based on the "Vee" diagram, illustrated in Figure 1.4.

Figure 1.4 shows the stages of building a system, with a symbolic "V" showing the progression from the top of the left leg of the "V" down to the base, across the base, and up the right leg. The project definition stages down the left side begin with development of a Concept of Operations, continue with Requirements and Architecture, and Detailed Design. The Implementation stage is shown across the base of the "V", with an arrow labeled "Time" pointing right to left across the bottom of the "V". The right leg shows the testing and implementation stages of a system, with an upward-pointing arrow showing the progression from the base up the leg. [11]

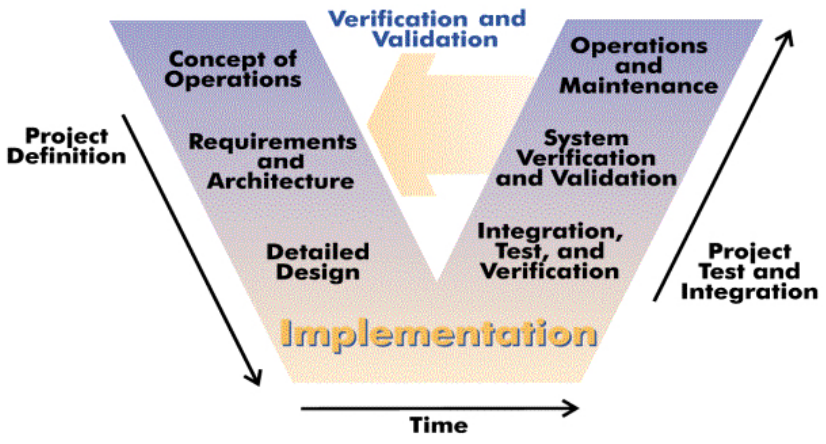
Verification and Validation
Concept of Operations
Operations and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Figure 1.1 The traditional V model [11]

The V-Model is the basic scheme of a software development. In the left leg, we proceed down by decomposing the specifications into architecture level design, and the architecture design into component level designs. Every level of decomposition have it's own specifications. After the implementation process, we verify every levels of decompositions behavior with it's specification.

1.7 Hierarchical runtime verification

A hierarchical runtime verification is based on the communication of runtime monitors with a higher level logic instead of a one time verification in design time. By using the V-Model, the right legs verification steps can be replaces with its runtime verification counterpart (Figure 1.2). These replaced elements are:

- Component monitors which are the smallest, can be implemented into embedded devices.
- Architecture monitors which are verifying the interoperation of components.
- High level logic is a central element for reacting to the states of the monitors.

1.8 Model driven software development

Model driven software development (MDSD) emphasizes problem solving by the development and maintenance of models describing the system being designed. MDSD

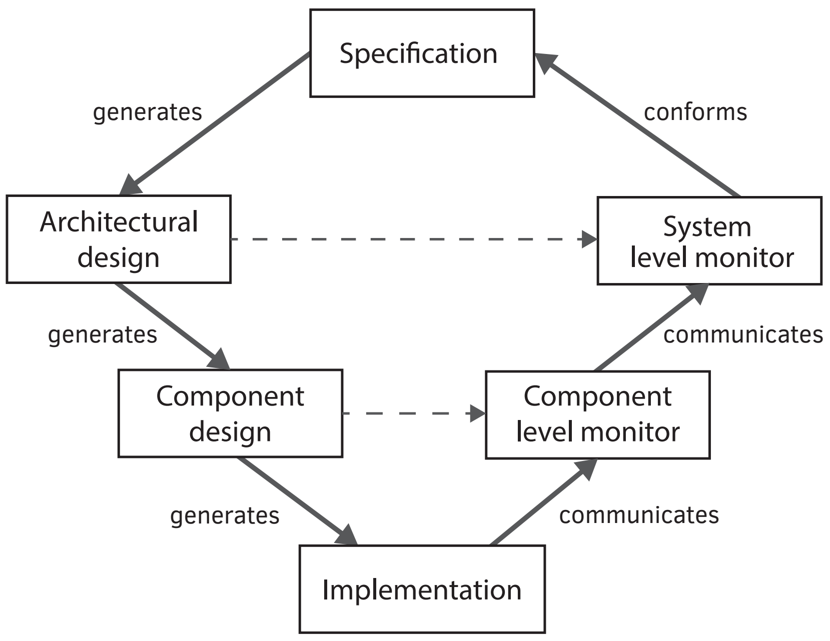

Figure 1.2 Hierarchical verification based on the V-Model

heavily relies on automated code and documentation generation based on the models of components or the overall model of the system.

Modeling has the advantage of introducing abstractions, thus reducing the complexity of the development process. The dynamic code generation guarantees that the code will inherit the properties that can be directly derived from the model, while reducing the costs by eliminating unnecessary round-trip engineering. The generation of documentation also results in an always up-to-date description of the components, stored together with the requirements and the model. Furthermore, model based approaches have the advantage of easier testability, or if the model is formal enough they can make verification possible.

Formal verification is especially important for the development of safety critical systems (e.g.: space and flight technologies), making MDSD notably widespread in these areas.

Various methods and tools are available for the generation of test cases and monitoring components from models, as well as for formally verifying certain properties. These tools usually support a few modeling formalism of their target domain.

MDSD is usually accompanied by the Y model – a software life cycle model for component based systems [3].

1.9 Y model

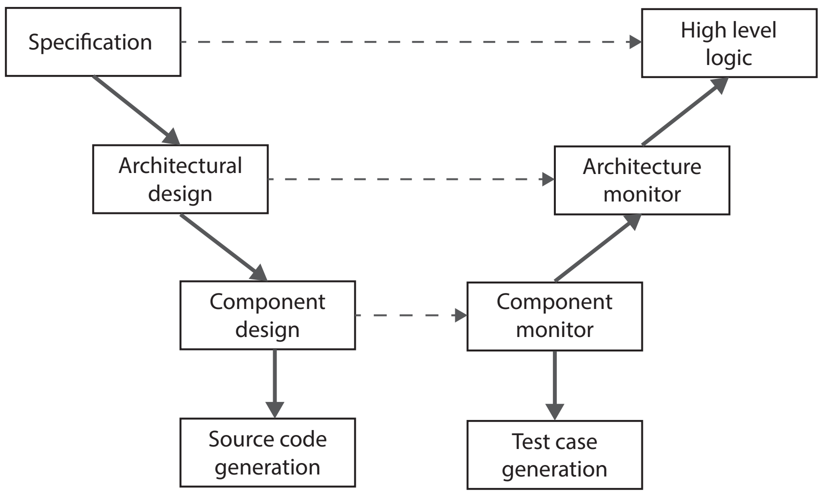

Figure 1.3 The Y model

The Y model [3] is an extension of the V model (which is in turn an extension of the waterfall model) [14], by using code and test case generation. Much like the V model, the development process is partitioned vertically. Each level contains a model that is transformed to a verification model, on which formal methods can be applied. The results of the verification process can be traced back to the original models making iterative improvement possible. The top level is for high level system models, while the second level contains architectural models, and the third one is for component based models. This provides input for the last step, source code and configuration generation for the individual components. Test cases are paired with the source code and can be generated from the component verification models.

1.10 Modeling approaches

MDSD methods require modeling languages to describe the behavior of systems and components. Engineering practices developed a wide range of such languages over the years to support fast paced product development. This allows the use of domain specific languages, which leads to a shorter modeling process but challenges formal verification software, as their input is usually stricter and in a more general format. The result is the need for complex model transformations before the verification can begin, which can lead to higher development costs, or – if the transformation contains errors – even faulty behaviour.

As a result, standardized modeling languages were developed like the UML (Unified Modeling Language [2]) and SysML (Systems Modeling Language [1]) languages.

1.10.1 CEP

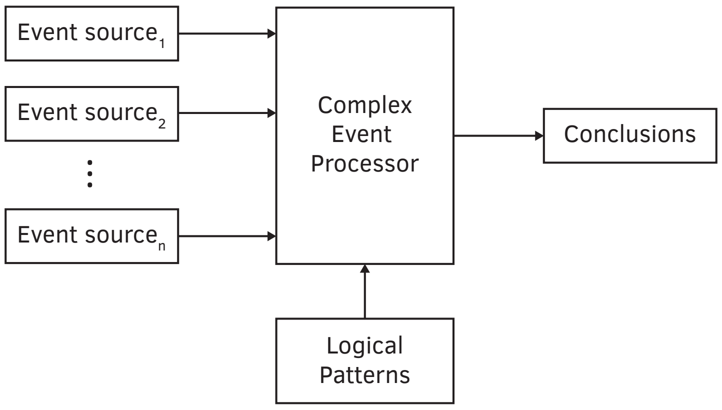

Figure 1.4 CEP overview

Complex event processing is a method of tracking and analysis streams of information and deriving conclusions. In a complex event processing environment, there can be multiple event sources, and with logical patterns given by a formalism, we can find patterns in the incoming stream, e.g. events followed by another events in some sequence: [**epbas**]

1.10.2 Finite automaton

Modelling a system with finite state space is often done by using finite automata – also known as finite state machines. A finite automaton accepts a (finite) list of symbols and produces a computation of the automaton for each input list. Although finite automata can be easily visualized, this formalism describes a simple, flat transition system and lacks the support for higher level concepts. The development of finite automata models are supported by many tools (e.g.: Finite State Machine Designer [10]).

1.10.3 Statechart

Statecharts, also known as state machines are an extension of finite automata. There are multiple available syntaxes for statecharts (e.g. the one defined by UML [4]). The higher level concepts that were introduced include variables, actions, and hierarchically nested states. Event-driven execution is also possible by using signals as the triggers

of transitions. Available variable types heavily depend on the concrete semantics of the chosen statechart language. Actions can usually be variable assignments, signal raises, or the setting of timers. Hierarchy lets users organize system descriptions using a top-down approach. Support for hierarchy is introduced via nested states and parallel regions. States can also have entry and exit actions, which allows the description of common functionality in parent states [12].
Statecharts are usually created by tools that support the graphical design of the model (e.g.: Yakindu, an Eclipse based editor).

1.10.4 Sequence Chart

The formalism of message sequence charts (MSC) describes the communication between components – the order in which messages can occur [7] [5]. The message interchange is usually represented by a graphical model. These charts can be used for high level specification, design, trace based testing, or documentation. A collection of possible sequence charts can also describe a complete communication protocol between components. UML sequence diagrams were inspired by MSCs, but their semantics differ regarding some of the basic elements of the language such as lifelines and arrows [6].

1.11 Contributions

Our goal was to provide runtime verification methods for all levels of a complex cyber-physical system. These methods should rely on modelling methods already being used be engineers to minimize the steepness of the learning curve and allow quick and efficient development. The result is a hierarchical runtime verification framework which can enable verification on two levels (Figure 1.5) The framework is capable of:

- Component level verification as monitors generated from statechart formalism
- System level verification, that can be done by a complex event processing system
- Monitors can generate messages indicating erroneous operation of the monitored components
- The complex event processing system can use the monitors messages

Additionally, the statechart language developed extends the usual statechart formalism by adding support for statechart templates. This enables users to easily describe systems with homogeneous components. The models can also be mapped to a formally verifiable transition system.

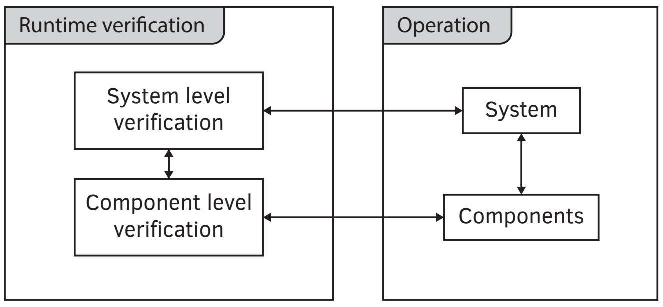

Figure 1.5 Associations between operation, and runtime verification components

With component level verification – in out case with embedded monitors – we can verify each component runtime state. The system level verification observes the overall state of the system. If any of the components reports failure, we can make a decision based on the remaining components abilities to support the system verification. If we can provide safety despite component failures, the system can continue operation.

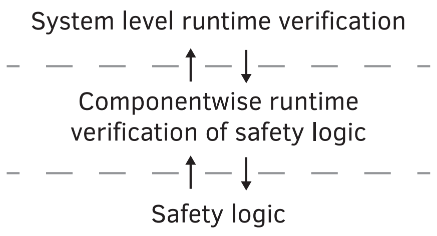
System level runtime verification
Componentwise runtime verification of safety logic
Safety logic

Figure 1.6 Overview of the hierarchical runtime verification system

It's important to notice that our solution is not tailored to this special problem but it is a general approach for any critical system. Our main goal was to develop a method which can integrate multiple safety logic into a global runtime verification, increasing the reliability of the complete system.

References

[1] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[2] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. “OMG unified modeling language specification”. In: *Object Management Group* 1034 (2000), pp. 15–44.

[3] Luiz Fernando Capretz. “Y: a new component-based software life cycle model”. In: *Journal of Computer Science* 1.1 (2005), pp. 76–82.

[4] OM Group et al. “OMG Unified Modeling Language (OMG UML), Superstructure”. In: *Open Management Group* (2009).

[5] David Harel and PS Thiagarajan. “Message sequence charts”. In: *UML for Real*. Springer, 2003, pp. 77–105.

[6] Øystein Haugen. “Comparing uml 2.0 interactions and msc-2000”. In: *System Analysis and Modeling*. Springer, 2005, pp. 65–79.

[7] Øystein Haugen. “MSC-2000 interaction diagrams for the new millennium”. In: *Computer Networks* 35.6 (2001), pp. 721–732.

[8] John C Knight. “Safety critical systems: challenges and directions”. In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE. 2002, pp. 547–550.

[9] James N Martin. “Overview of the EIA 632 standard: processes for engineering a system”. In: *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*. Vol. 1. IEEE. 1998, B32–1.

[10] Andreas Muelder. “Yakindu”. In: *Jun-2011.[Online]. Available: http://www. yakindu. org/yakindu/.[Accessed: 20-Aug-2009]* (2011).

[11] Leon Osborne, Jeffrey Brummond, Robert D Hart, Mohsen Zarean, and Steven M Conger. *Clarus: Concept of operations*. Tech. rep. 2005.

[12] Miro Samek. “Who moved my state”. In: *Dr. Dobb’s Journal* (2003).

[13] Kenneth J Schlager. “Systenas Engineering-Key to Modern Development”. In: *IRE Transactions on Engineering Management* (1956).

[14] Seema Suresh Kute and Surabhi Deependra Thorat. “A Review on Various Software Development Life Cycle (SDLC) Models”. In: *IJRCCT* 3.7 (2014), pp. 776–781.

[15] Dolores R Wallace and Roger U Fujii. “Software verification and validation: an overview”. In: *IEEE Software* 3 (1989), pp. 10–17.