Executable Specifications: Language and Applications

Dr. Doron Drusinsky and Dr. J. L. Fobes
Naval Postgraduate School

With recent industry focus on software safety and dependability, and with a particular Department of Defense (DoD) emphasis on safety-critical systems, formal methods are gaining renewed attention as a way to ensure that an implementation is consistent with its specifications. Unlike conventional testing methods, which are human intensive and therefore slow, expensive, and error-prone, formal methods enable automated computer-aided verification. This article describes an easy-to-use formal method based on executable specifications. In particular, it describes the logical foundation of executable formal specifications, along with some interesting applications relevant to DoD missions such as run-time monitoring of real-time systems and online temporal rule checking for security-based applications. It also describes two categories of techniques for executing formal specifications.

Formal specification languages have been thoroughly investigated during the past three decades. They have been considered primarily for verification and validation purposes using techniques commonly known as formal methods. Most formal methods have suffered from limited commercial success due to several limiting factors such as their prohibitive computational complexity and the high level of mathematical skills needed to be used effectively.

Recent research has focused on *executable specifications*, a new class of applications of formal specifications whereby specification rules are executed on a computer much like any high-level programming language. This class of techniques and associated tools harnesses the linguistic power of formal specification languages yet is simple and does not suffer from the complexity limitations of formal verification methods. In addition, executable specifications enable new application domains in addition to classical verification such as online temporal reasoning in security applications.

In this article, we focus on temporal logic that is a particular and prominent formal specification language. We begin with background information about logic and formal methods and then describe temporal logic in greater detail. Next, we describe executable specification methods and tools followed by a description of a successful verification effort using executable specification. Lastly, we describe security rule-checking using executable specifications.

Background

Formal specification languages are designed to capture requirements (what a system should do) in a formal way, i.e., using mathematics. In contrast, design and programming languages capture the implementation (*how* a system implementation does what it is supposed to do).

Using mathematical notation to capture specifications removes potential ambiguity and, when coupled with mathematical proof techniques, enables program correctness proofs. These proofs provide indisputable statements about the absolute absence of errors in the implementation. This contrasts with testing techniques where only incomplete evidence is provided. The body of knowledge involving formal specifications and formal correctness techniques is commonly referred to as *formal methods*.

> *"Using mathematical notation to capture specifications removes potential ambiguity and, when coupled with mathematical proof techniques, enables program correctness proofs."*

Clearly, there is an inherent trade-off between investing in the education and tools of formal methods versus the potential benefit of assuring bug-free software. For example, a low-end Web site owner might be willing to take the risk of having program errors on the site rather than invest in costly verification methods. On the other hand, the cost of a single bug in the software on board a multimillion dollar space mission justifies the investment in robust verification techniques such as formal methods.

The most popular mathematical domain used by formal specification languages is logic. In its simplest form, Boolean *propositional logic* [2] is the kind of logic found in every modern programming language such as the C/Java expression (x>0) && (y==1). However, propositional logic is not powerful enough to elegantly capture temporal and aggregational aspects of the system. For example, propositional logic cannot explicitly state that (x>0) *must be true now* and (y==1) *must be true sometime within the next 5 seconds.*

First Order Logic (FOL) extends propositional logic with two quantifiers: the universal quantifier (∀ read as *for all*), and the existential quantifier (∃ read as *there exists*). These quantifiers range over a known set, i.e., the set of all cars registered in California. Hence, a statement such as *a California registered minivan must be at most 10 feet long* can be stated in a single expression: ∀car: minivan → (length≤10ft.).

In contrast, using propositional logic would require that you explicitly state – for every car in the set – the above statement. Also, using programming techniques to achieve the desired aggregate effect defeats the whole purpose of specification, i.e., to make a clear statement about *what* the system should do without dealing with the *how* it does so.

Linear-Time Temporal Logic (LTL), the formal specification language described later in this article in the section "REM Tools: Code Generators and Monitors," extends propositional logic with four temporal operators. LTL has an advantage over FOL in that it removes mathematical clutter and enables specifications in a form that is close to natural language. It is mostly suitable for *reactive systems*, i.e., systems that constantly interact with their environment such as control software in a cruise missile.

Two primary classes of formal correctness proof techniques are theorem provers and model checkers. Theorem provers use

logic proof methods to prove that a program conforms to a given specification. Theorem provers support only a subset of LTL, and typically require a highly skilled human driver. Model checkers, on the other hand, are automatic and support full-blown LTL specifications (though typically with little support for real-time constraint validation). However, due to their prohibitive computational complexity, model checkers tend to work well only for small programs.

Run-time Execution and Monitoring (REM) is an effective and efficient hybrid between formal methods and conventional execution or simulation-based testing techniques. REM uses LTL-based specifications augmented with real-time constraint specifications. REM is a method of automatically comparing the behavior of an underlying application such as an embedded system to its formal specification. This is done by executing the specification in tandem with the application.

While REM uses formal specifications, it is not a pure mathematical proof technique; test-based verification and corresponding test suites are still required. Nevertheless, REM is simple to use and automates the verification process. In addition, REM tools described in the sequel are capable of detecting real-time requirement violations while executing on an embedded target. Interestingly, REM is also useful for non-verification applications such as security checking, as described in the "Security Applications" section of this article.

Formalism and Language Lessons

LTL is an extension of propositional logic that deals with time and order. As early as 1977, LTL was proposed as a way to formally specify multithreaded programs [1, 2[1]]. Since then – and especially during the last decade – researchers have expanded its theoretical and practical power using it, for example, to specify protocols and hardware. LTL is a simple and intuitive extension of propositional logic that is closer to natural language than most other specification languages. For those reasons, LTL is the formal specification language used by most formal methods and is also the specification language of choice for REM methods and tools.

The syntax of LTL adds eight operators to the AND, OR, IMPLIES, XOR, and NOT of propositional logic. Four of the operators deal with the future: Always in the future, Eventually (sometime in the future), Until, and Next cycle; additionally, four dual operators address the past: Always in the past, Sometime in the past, Since, and Previous cycle.

Metric Temporal Logic (MTL) enhances LTL's capabilities [3]. With it, you can define upper and lower time constraints as well as time ranges for the LTL operators. By imposing relative and real-time constraints on LTL statements, MTL lets you use LTL to verify real-time systems. The following is an example showing a relative-time upper boundary in MTL:

Always$_{<10}$(readySignal Implies Next ackSignal)

which reads,

> Always, within the next 10 cycles, readySignal equals 1 implies that one cycle later ackSignal equals 1. The text inside the parentheses is a propositional logic expression, and a cycle is an LTL time unit, which is a user-controlled quantity. The time constraint is relative in that it is counted in clock cycles.

Another example is as follows:

Always $_{\text{timer1}[5,10]}$(readySignal Implies Eventually $_{\text{timer2}>=20}$ ackSignal)

which reads,

> Always, between 5 and 10 timer1 real-time units in the future, readySignal equals 1 implies that eventually, at least 20 timer2 real-time units further in the future, ackSignal equals 1. Here, two real-time constraints are specified using timer1 and timer2 clocks. A separate statement maps these timers to system calls, system clocks, or another counting device.

One reason for the prominence of LTL as a specification language is the simple way in which it relates to natural language. For example, consider the natural language requirement for a traffic light controller (TLC): *whenever light is red, light should turn green within two minutes*. The following conversion steps convert the requirement into an MTL requirement.

1. Always when light is red then light should turn green within two minutes.
2. Always (if light is red, then light should turn green within two minutes)
3. Always (light is red implies that eventually light should turn green within two minutes)
4. Always (LightColor==RED Implies Eventually $_{\text{seconds}<120}$ LightColor== GREEN)

REM Tools: Code Generators and Monitors

The two primary categories of executable specification tools are code generators and REM tools (monitors). Code generators generate source code in a programming language such as a Java, C or C++, from formal, LTL, specifications. While a conventional program can handle propositional logic[2], it cannot deal with higher forms of logic such as FOL, LTL, or MTL. For example, writing LTL inside a C program will result in compilation errors. Therefore, an often-used solution embeds high-level specification requirements inside program comments.

For example, the following C program contains an embedded MTL assertion for a TLC (written with syntax from [4]) asserting that *for 100 milliseconds, whenever light is red, camera should be on*:

```
void tlc(int Color_Main, boolean
      CameraOn) {
… /* Traffic Light Controller
      functionality */
/* TRBegin
TRClock{C1=getTimeInMillis()} // get
      time from the OS
TRAssert{ Always({Color_Main ==
      RED} Implies
 Eventually_C1<1000_{CameraOn
 == 1})
 } =>
// Customizable user actions
{printf("SUCCESS\n");printf("FAIL\n");
      printf("DONE!\n");}
TREnd */
} /* end of tlc */
```

An executable specifications code generator generates code that replaces the embedded LTL/MTL assertion with real C code, which executes in process with the rest of the TLC, i.e., as part of the underlying TLC application. The generated code can also be used for formal specification-based exception handling [5].

In contrast to code generators, REM monitors (e.g., [6, 7]) monitor assertions in a stand-alone process often on a remote machine. It uses Hyper Text Transfer Protocol (HTTP), sockets, or serial communication to interface with the client application. To monitor, these tools either generate special, out-of-process source code using a code generator or use mathematical tools such as rewriting systems. The following list describes desired properties of remote monitors:

1. **Monitoring online, namely, no post-mortem processing is used.** A counter example would be to store all events in a database and use a structured query language (SQL)-based method to query those tables at a later time. The motivation for this requirement is that no expected termination time for the underlying application (e.g., security application) should be assumed. With no expected termination time, the size of the stored history trace information will be monotonically increasing and unbounded, which is unacceptable in most cases.
2. **Low impact**. It is desirable for a REM tool to not actively interrogate the client application (e.g., the banking system in the R2 example in the "Security Applications" section). Rather, it should passively listen to an event stream pertaining to basic propositions such as *deposit-occurred* or *balance<0*, which is sent to the REM tool from the client application. Having a low-impact REM tool increases the likelihood of acceptance by commercial and security-related organizations. For example, a bank is typically unwilling to be actively interrogated by a third-party tool but might agree to voluntarily, on its own terms and conditions, send out limited information to a third party such as a REM monitor.
3. **Powerful and flexible rules language.** Formal specifications need to capture real-life patterns and concerns such as real-time constrains while being syntactically close to natural language. The LTL satisfies these requirements; a large body of research points to its expressiveness and usefulness as a specification language. The MTL adds real-time constraints to LTL specifications. Time series constraints are also supported [8].

Run-Time Monitoring of Safety Critical Systems at NASA

NASA's Jet Propulsion Laboratory has used REM to verify the fault protection subsystem of the Deep-Impact spacecraft [9]. The level of fault protection provided by this system is single-fail-operational (the system has the capability to recover from a single fault and continue its mission). Multiple faults are handled sequentially, where only one response is active at a time.

The fault protection provides monitoring and system-level responses to faults detected onboard the spacecraft. Other missions that have used this level of fault protection include Galileo, Mars Pathfinder, and DS1. Deep-Impact continues with this legacy but has incorporated a core reusable portion called the Fault Protection (FP) engine. The FP engine is responsible for accepting all input symptoms from various monitors and generating the appropriate system level response. These responses may vary from a single command (reset a hardware device) to a long running sequence (orient the spacecraft to a safe altitude).

The FP engine handles faults by priority-based input queues. The FP engine ensures that responses are handled in an orderly fashion and are not run unnecessarily (triggered by an overly sensitive monitor). After each response is run through to completion, the FP engine clears the fault and sends a cleanup signal back to the offending monitor.

REM was used to validate the FP engine software while in the development phase in executable form although not mature and robust; this led to the possibility of uncovering latent bugs in the software. Many aspects of the FP application exist that lend themselves well to a runtime verification approach. For example, during runtime the developer/tester may be unaware of inconsistencies within the internal state of the FP engine. Faults may be locked into a state where they are unable to trigger a response. Due to the nature of the application, there are hundreds of possible symptoms that may be reported to the FP engine, and dozens of possible faults that can trigger responses. It would be difficult for a test engineer to be constantly checking internal state consistency. Using the runtime verification approach, the REM monitor executed in parallel with the application, alerting the user to any violation of pre-defined correctness properties.

"Formal specifications need to capture real-life patterns and concerns such as real-time constrains while being syntactically close to natural language."

Security Applications[3]

Consider the following two airline security-related temporal pattern rules, both concerned with detecting a foreign national male passenger with a student visa flying to the Harrisburg International Airport near the Three Mile Island nuclear power plant:

- R1. Detect such a passenger if he has traveled to the Middle East at least once within a year of obtaining his student visa.
- R2. Detect such a passenger if he has traveled to the Middle East at least once within a year of obtaining his student visa and he received two or more direct deposits from non-US banks within the last year.

Both rules describe temporal patterns that contain potentially discernable elements from an airline security system operating automatically and in real-time. The two primary methods for performing such temporal pattern detection are *offline* and *online*, as described in the sequel.

The Federal Aviation Administration has an automated profiling system originally termed Computer Assisted Passenger Screening (CAPS) [10] that relies upon the data in each Passenger Name Record. This profiling system is being upgraded to access a more extensive range of data. The upgrade, CAPPSII, will profile airline passengers based on secret criteria to identify potential terrorists. Personal information about passengers may additionally include that from the Immigration and Naturalization Service (INS, now U.S. Immigration and Customs Enforcement), law enforcement, and customs. Having such history information stored in the system, or in constituent subsystems, enables SQL-based implementation of a temporal pattern rule such as R1. We call such an implementation *offline* because it relies on storing and querying historical information. An offline solution induces the following three impact consequences:

1. Temporal historical information is stored within the system (e.g., within CAPPSII and/or its constituent subsystems).
2. Temporal and non-temporal pattern detection is initiated by the security-related query, querying the constituent resources at will. We regard this as a high-impact solution because a temporal query is initiated from outside the original scope of the queried system, thereby impacting the performance of the queried system. For example, a potential INS subsystem of CAPPSII being impacted by repeated external queries from CAPPSII proper which, sooner or later, will degrade the INS system performance. Performance

degradation will occur because of the actual query processing forced on the INS subsystem and because of the fact that as time progresses, temporal queries might query monotonically increasing data-sets of historical data.
3. In addition to performance issues, CAPPSII and its constituent subsystems need to agree on a shared data representation for merging query results from multiple subsystems (e.g., merging INS and law enforcement query results).

This article is concerned with *low-impact*, *online* temporal pattern detection. It uses REM to detect temporal patterns without using historical data (i.e., it is online), and without querying the underlying application (i.e., it is low-impact). The only communicated information it requires from the underlying application (e.g., CAPPSII constituent subsystems) are Boolean messages for basic propositions such as *deposit of more than $1,000 was made to account of SSN=222 11 2222*.

While pattern rule R1 described earlier is programmable within the suggested CAPPSII framework, R2 requires extension, which includes banking information. Such an extension however will not lend itself to offline temporal pattern detection methods for the following reasons:

1. The banking data systems only store historical/temporal information for a limited duration (e.g., three months). The industry is unlikely to make any significant change to this policy.
2. Banking data systems are not likely to permit high-impact, CAPPSII-initiated queries because of the performance consequences discussed earlier as well as their own security need to be in full control over any content query.

In contrast, a REM temporal pattern detection method, being online and low-impact, can be used in tandem with CAPPSII while supporting extensions that support rules such as R2.

Conclusion

Executable specification methods have been effectively used to verify safety-critical systems at NASA. They enjoy the power and accuracy of formal specifications, yet are easy to use. They also enable requirement simulation prior to implementation. In addition, these appealing properties of executable specification methods lend themselves to non-verification applications such as monitoring temporal rules within security applications.◆

References

1. Manna, Z., and A. Pnueli. "Verification of Concurrent Programs: Temporal Proof Principles." Proc. of the Workshop on Logics of Programs. Springer Lecture Notes in Computer Science Vol. 1981: 200-252.
2. Pnueli, A. The Temporal Logic of Programs. Proc. of 18th IEEE Symposium on Foundations of Computer Science, 1977: 46-57.
3. Chang, E., A. Pnueli, and Z. Manna. Compositional Verification of Real-Time Systems. Proc. of 9th IEEE Symposium on Logic in Computer Science, 1994: 458-465.
4. Drusinsky, D. "The Temporal Rover and ATG Rover." Proc. of Spin2000 Workshop. Springer Lecture Notes in Computer Science Vol. 1885: 323-329.
5. Drusinsky, D. "Formal Specs Can Handle Exceptions." CMP Embedded Developers Journal Nov. 2001: 10-14.
6. Drusinsky, D., and M. Shing. Verification of Timing Properties in Rapid System Prototyping. Proc. of Rapid System Prototyping Conference, 2003.
7. Havelund, K., and G. Rosu. Monitoring Programs Using Rewriting. Proc. of IEEE Conference on Automated Software Engineering, 2001.
8. Drusinsky, D. "Monitoring Temporal Logic Specifications Combined With Time Series Constraints." The 4th Joint Workshop on Formal Specifications of Computer-Based Systems, 2003, extended abstract.
9. Drusinsky, D., and G. Watney. "Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine." The 27th IEEE/NASA ICECCS Workshop, 2002.
10. Fobes, J.L. Computer Assisted Passenger Screening (CAPS), DOT/FAA/AR-96/38, 1996.
11. Drusinsky, D., and J.L. Fobes. "Real-Time, On-Line, Low-Impact, Temporal Pattern Matching." The 7th World Multiconference on Systemics, Cybernetics, and Informatics, 2003.

Note

1. Published in [11].
2. Propositional logic is a mathematical model that allows us to reason about the truth or falsehood of logical expressions.
3. Professor Amir Pnueli, a longtime advocate of temporal logic, is the 1996 Turing award winner for his "seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification."

About the Author

Doron Drusinsky, Ph.D., is an associate professor of computer science at the Naval Postgraduate School, Monterey, Calif. His primary research interests are run-time monitoring and verification of safety-critical software systems. Drusinsky worked for Sony from 1988 until 1993 when he founded R-Active Concepts and authored BetterState, a UML statecharts design tool. BetterState was acquired by ISI-WindRiver Systems in 1997. Doron established Time Rover, Inc. and authored the Temporal Rover and DBRover formal verification tools. He has a Bachelor of Science from the Technion, Israel, and a doctorate from the Weizmann Institute.

Code CS/Dd
Naval Postgraduate School
Monterey CA, 93943
Phone: (831) 656-2168
Fax: (831) 656-3225
E-mail: ddrusins@nps.navy.mil

J.L. Fobes, Ph.D., conducts research and development for the U.S. Department of Homeland Security's Transportation Security Administration while serving at the Naval Postgraduate School, Monterey, Calif., as the Transportation Security chair. Formerly he was an associate professor at California State Los Angeles, conducted research for the U.S. Army Research Institute, was the chief of Human Factors for the U.S. Army Operational Test and Evaluation Agency, and the Federal Aviation Administration's program director for Aviation Security Human Factors. Fobes has a doctorate from the University of Arizona and a post doctorate from the California Institute of Technology, Pasadena.

Naval Postgraduate School
Monterey CA, 93943
E-mail: jlfobes@nps.navy.mil