# CrossCheck: Improving System Confidence through High-Speed Dynamic Property Checking

Jonathan Springer, James Ezick, David Wohlford
Reservoir Labs, Inc.
{springer, ezick, wohlford}@reservoir.com

Matthew Craven, Rick Buskens
Lockheed Martin
{matthew.craven, rick.buskens}@lmco.com

## Summary

Increasingly, the time to deploy a complex system is being dominated by the complexities of generating, managing, reasoning about, and ultimately validating immense amounts of information. Lengthy prose specifications impose high-level constraints which may transcend reasoning about a single component. Components developed in isolation, to shifting design requirements, must interoperate. Rigorous validation requirements, such as those for avionics imposed by RTCA/DO-178B, mandate documented proof of comprehensive test coverage. Even after deployment, there remains a need for logging and analysis of nominal use and the ability to maintain system confidence through random failures and variable environmental factors.

To address these needs, we have developed the CrossCheck™ system for dynamic property checking and recovery. CrossCheck has two key assets:

1. **CrossCheck abstracts the checking problem:** Specification patterns are defined in terms of abstract "events" which are programmer-specified blocks of structured data. CrossCheck is not tied to any particular transmission medium but rather, through the implementation of a simple interface, can accept event streams from any source, for example via DDS [1].
2. **CrossCheck provides high-speed and high-throughput checking**: The technology behind the CrossCheck checking engine is motivated by algorithms from modern network intrusion detection appliances. Complex operations can be expressed in ANSI C and compiled to native code rather than being interpreted. The structure of the specification language makes it possible to avoid constructing state-machines that are exponential in the size of the property being checked.

We have a development implementation that includes the checking engine and a compiler for the CrossCheck Specification Language (CSL). CSL specifications are hierarchical and include native code handlers for hooks generated by the checking engine at events such as successful expression matching and rule-initiated recovery. Our development system also includes a collection of event-generation drivers that have allowed us to experiment with a range of applications.

## Application Domains

CrossCheck is a general-purpose dynamic checking system applicable to a wide range of application domains. The principal requirement for applying CrossCheck to an application (whether a program, a control system, or other similar item) is that it be amenable to formulation of high-level properties expressed in terms of event abstractions. CrossCheck is principally useful when applied to such systems whose operation is too complex to be fully characterized statically.

Scenarios CrossCheck was developed to address include:

**Online Verification of a Flight Control System**

Components such as navigation units (e.g. an Embedded GPS/Inertial (EGI) navigation system), onboard sensors, and mission planning units may emit streams of data that can be marshaled into CrossCheck events. Specifications can be written to enforce rules from basic safety properties to complex mission sequencing requirements.

**Systems Integration Checking**

Open architecture frameworks such as the Software Communications Architecture (SCA) used by JTRS [2] for developing software radio products and software libraries such as VSIPL used by signal processing developers define contracts on usage and interoperability between components. CrossCheck can enforce these contracts by monitoring at the boundary between interacting components.

**Fault Injection for Automated Coverage Testing**

CrossCheck supports event feedback to the system being checked when a rule is matched. Specifications can track system progress, allowing specific conditions to trigger targeted modification of the system environment. Scripts of these modifications can facilitate coverage documentation of rare cases or direct injection of faults to test behavior.

**Checking of Automated Planning Systems**

Autonomous systems often rely on opaque and highly complex systems to achieve sophisticated behavior. For these systems, static analysis is typically not possible. An independent, dynamic checking framework can be invaluable as a safeguard in these cases, especially when the cost of failure is high. We have experimented with the cognitive application framework Soar [3].

**Deep Network Protocol Inspection**

Sophisticated attacks may subvert the implementation of transport-level or application-level protocols. Traditional signature matching techniques are inadequate in these cases, but by relating protocol states to CrossCheck events, protocol misuse can be detected. CSL can express advanced protocol specification languages such as binpac [4].

This work was supported by Air Force Research Lab, under Phase II SBIR contract FA8750-07-C-0049.

## CrossCheck Architecture

A complete CrossCheck use case consists of the CrossCheck runtime and compiled specification together with an event generating application (Figure 1). The CrossCheck user defines a set of events relevant to the application domain and a specification for their behavior.

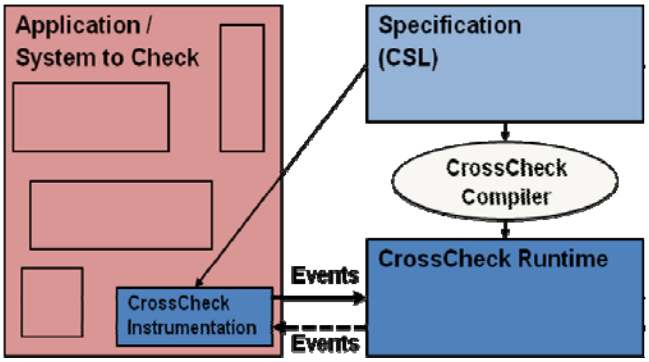


**Figure 1: CrossCheck System Diagram**

## Runtime Checking System

Our algorithm for checking is based on rendering much of the specification as C code whose execution is initiated by an abstract machine processing CSL operators. Our CSL compiler performs this transformation and links the generated C code against a runtime library to produce a specialized, fully-compiled checking engine, accepting a stream of events from the application and reporting property violations. This approach maximizes performance and provides good platform portability with a low footprint.

On receiving an event, the checking engine determines the set of applicable rules and orders their evaluation according to a partial order that can be stated in the specification. Rule match states are advanced, with new independent match lines being forked and dead lines involving match failure being deleted. Each line maintains its own variable bindings supporting robust rule templates. A rule match initiates a "recovery" operation that may or may not modify the current event. Modification rolls back the checking state to the beginning of the first applicable rule. This iteration continues until all the event "falls through" all of the applicable rules and is emitted for logging or return to the system under check.

## CrossCheck Specification Language

Specifications are written in the CrossCheck Specification Language (CSL). CSL is a general-purpose specification language based on the theory of nondeterministic context-free languages, extended with the ability to refer to arbitrary functions in key places. Specifications are described in terms of hierarchical productions, which retain the flavor of the traditional representation of context-free languages. These productions operate over the stream of events coming from an event source (e.g. the system under check). For expressivity, CSL productions support bounded Kleene operators and propositional connectives inspired by familiar regular expression syntax. The use of regular expression-style operators facilitates efficient execution for simple patterns, while the ability to refer to general-purpose code gives flexibility to implement very general context-sensitive matching that can include deep reasoning and analysis.

Key to the versatility of CrossCheck is the interplay between CSL and the checking engine. The engine provides facilities such as simple transactional data storage that can be used to hold global data. These facilities are exposed to CSL rules through an API that can be called from the compiled code fragments. In turn, the checking engine has hooks at various processing points to which CSL defined handlers can be attached. These hooks allow the checking problem to be abstracted while the runtime provides facilities that make powerful specifications easy to write.

## Example from Sensor Processing

In a flight control scenario, it may be desirable to report any situation in which acceleration exceeds some maximum force extending over a given period of time. The system under check in this case is assumed to have a navigation sensor that reports acceleration force at regular intervals.

A partial CSL specification is given in Figure 2. This specification defines an event format that includes acceleration readings in three dimensions. These values would be intercepted by CrossCheck when transmitted over the on-board messaging bus. (The CrossCheck runtime supports TCP/IP natively.) A series of predicates detect acceleration that exceeds a specified maximum in any one of these dimensions. The `AccelHigh` predicate checks that such acceleration is not maintained for too long (three time steps in this case), using regular expression-style alternation and iteration (bounded Kleene-star) operators.

```
NAV(x_acc:double, y_acc:double, z_acc:double) ;;

%%

HighXAccel <- <x_accel_high>? ;;
HighYAccel <- <y_accel_high>? ;;
HighZAccel <- <z_accel_high>? ;;
AccelHigh  <- (HighXAccel | HighYAccel | HighZAccel)*[3] ;;

AccelRule := AccelHigh,
  <rule_index_f_all>, <rule_init_f_nop>,
  <rule_rec_f_nop>, <rule_destroy_f_nop>, 0,
  (oldest_only, no_rollback), (),
  "Acceleration too high for three timesteps" ;;

%%

DECLARE_PREDICATE_P(x_accel_high, e) {
  return (e->type == ET_NAV && e->u.nav.x_acc > 0.5);
}
```

**Figure 2: High acceleration specification fragment**

The predicate that detects high x-acceleration is shown; the others are similar. Through the programming interface, predicates can bind variables to values within the context of a rule match so that down-stream predicates can check for a matching instantiation. We are developing a library of common rule fragments and predicate templates. More complex forms can be written by hand or generated by tools specialized for the application domain.

## References


[1] http://portals.omg.org/dds

[2] https://jtel.spawar.navy.mil/

[3] J. Laird, A. Newell, and P. Rosenbloom, "Soar: An Architecture for General Intelligence," In *Artificial Intelligence*, v. 33, p. 1-64, 1987.

[4] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: A yacc for Writing Application Protocol Parsers," in *Internet Measurement Conference*, USENIX, Oct. 25-27, 2006.