

# High-Availability in $\delta$ -Core: A Formal Derivation

*This paper presents a formal specification of high-availability features underlied in the embedded real-time operating system kernel  $\delta$ -Core. The specification is described in terms of constructive type theory supported by a mathematical theorem proof development system PowerEpsilon. Many mathematical properties have been derived in the framework.*

## 1 INTRODUCTION

### 1.1 Demands for High-Availability

Smart devices for service-critical Internet infrastructure equipment and Internet appliances, as well as mission-critical defense and aerospace systems and life-critical medical products, must satisfy market demands for new, higher levels of reliability, availability, and serviceability (RAS). Initial approaches aimed at delivering RAS attributes, in systems referred to as having high availability (HA), originated in the telecommunications industry for its infrastructure equipment more than ten years ago. Neither flexible nor efficient, such legacy HA technology and programming paradigms fall far short of providing optimal solutions for today's fast-moving Internet-based marketplace.

Creating products that meet market criteria for high availability -- devices that operate 24 hours per day, 7 days a week, with 99.999 "five 9s") -- is challenging enough. Bringing those products to market on a timely, cost-competitive basis in the face of current market dynamics adds to the challenges and demands more efficient tools and programming methods. In today's market, software complexity and content continue to grow -- applications exceeding 1 megabyte of code are commonplace, as are large, often cross-corporation development teams. The isolated nature of embedded devices is breaking down as these systems are networked -- embedded platforms increasingly host "foreign" applications from independent software vendors.

As embedded devices transition from fixed functionality to open, multi-function managed appliances, embedded applications become increasingly dynamic as vendors allow customization (personalization), extensions, and software updates. Further, as our world economy becomes more service-based, its dependence on the time-lieness and availability of those services increases market pressure to avoid denial of service access caused by system downtime.

Taken together, these market trends create strong pressure to develop secure, fault-tolerant, manageable, serviceable systems. Given the likelihood that the networking of smart devices over the Internet will continue to increase rapidly, it is easy to forecast a requirement for pervasive high-availability capability within large segments of the embedded systems industry in

the not very distant future. Faced with this impending reality, it is time for the commercial off-the-shelf software (COTS) vendor community to define its role in providing HA solutions.

Until now, virtually all HA systems have been based on proprietary solutions. Efforts to create these HA systems have been hampered by rigid, inefficient protection models based upon Unix or Linux process models and message-passing operating system (OS) architectures. Among the drawbacks of such models are the inability to track and incorporate OS vendor upgrades and the difficulty (or impossibility) of integrating new technologies and third-party solutions into legacy HA systems. Further, proprietary systems incur the cost of in-house programming resources for their maintenance. The overhead of message-passing operating systems and the inflexibility of process models make these models inappropriate for creating tomorrow's HA systems.

A fundamentally new approach is needed. The core technology that COTS vendors can contribute is a truly HA-enabled operating system.

### 1.2 An Ideal HA Operating System

An HA-enabled operating system must enhance system uptime and operate in an environment in which hardware and software configurations are dynamically changing and can adapt. Conceptually, high availability can be viewed as a set of characteristics that enables a system to maintain a specified level of continued operation, even in the event of component failure or during hardware or software upgrades. No single system component can guarantee high availability. Rather, high availability depends on a careful combination of system software, specially designed hardware, management strategy, and application software techniques.

To provide for high availability, an OS has to deliver in all areas of reliability, availability, and serviceability. Additionally, it must also facilitate flexibility and adaptability in software development, promote code reuse, and offer scalability to match system resources.

Reliability within the context of HA has four components: application isolation, user/supervisor mode protection, resource management, and overrun protection.

- Application isolation means that software entities that are intended to operate independently do so and don't corrupt each other's code or data space.

Since applications that run in system or supervisor mode can execute privileged instructions that can do irrevocable damage.

- User/supervisor mode protection means that the OS must control the key scheduling and exception handling of the system so that applications run at a lower privilege level and do not have the kind of access that could compromise the integrity of the kernel code.
- In a dynamic HA environment, resource management, or reclamation, means the operating system must track, allocate, and clean up all the objects that make up the interaction of the overall system, like memory, tasks, queues and semaphores, as they are created, used, moved around, and destroyed.
- Effective overrun protection means that the OS uses limits, thresholds, and tracking methodology to protect itself from run-away applications that allocate too much stack or heap or that run at inappropriately high priorities.

An OS architecture for high availability calls for a built-in system for event management to provide a way to recover automatically from trapped exceptions and keep service running. That often requires a high level of system visibility to support external management of the system – the ability to go in and find out what is happening; to be made aware that errors are occurring, being recovered from, and so on. That, in turn, requires the ability to support the logging of events and the ability to go in and query those logs. It is time for the commercial off-the-shelf software (COTS) vendor community to define its role in providing high-availability solutions.

## 2 POWEREPSILON

**PowerEpsilon** [7, 8], is a strongly-typed polymorphic functional programming language based on Martin-Löf's type theory [4] and the Calculus of Constructions [2]. The system can be used as both a programming language with a very rich set of data structures and a metalanguage for formalizing constructive mathematics. The system has been implemented using the software development system AUTOSTAR constructed by author [5].

**PowerEpsilon** is a proof checker much similar to other mechanical proof checkers, such as LCF [3] and Nuprl [1], which are completely formal user-controlled systems. However, PowerEpsilon is more powerful than LCF and Nuprl, in which the equality and induction rules for arbitrary inductive types are definable. Since PowerEpsilon is composed of only a few constructs, it is a useful tool for studying and giving semantics to programming languages.

**PowerEpsilon** has been used for constructing a formal specification of an embedded real-time operating system called  $\delta$ -Core and a number of properties such as the safety, correctness, deterministics, and high-availability have been verified. Please see [6] for more details.

## 3 $\delta$ -Core

$\delta$ -Core is a 16/32-bit embedded real-time operating system.  $\delta$ -Core consists of the real-time kernel, a group of software components and the interface library (BSPs).

- **Device Driver Layer.** The device driver layer is constituted by device drivers, interrupt service routines (ISRs) and exception handling routines. It is the interface between the operating system and the hardware. This layer is independent of operating system and therefore provides strong portability of  $\delta$ -Core.
- **Real-Time Kernel.** The operating system kernel is the control center of the whole system. It provides real-time multitasking scheduling mechanism, supports the communication between tasks, synchronizes the real-time execution of the tasks, manages the interrupts in the system, provides the reliability and high-availability mechanisms, and therefore supports the users to develop safe and correct real-time systems.
- **Networking Software Components.** The implementation of this layer is in part dependent of the operating system kernel. It will also invoke the LAN card drivers in the device driver layer. Its functionality is to achieve the real-time communication over the tasks among the various of platforms interconnected on the network.
- **ANSI C Library.** This is a re-entrant ANSI C library building on the operating system kernel.

### 3.1 Features

As an embedded real-time operating system,  $\delta$ -Core has the following features:

- Fast and deterministic system responding time.
- Provides the functionalities of error checking and exception handling as well as the memory protection mechanism.
- Scalable and expandable software components.
- Most of code is developed using ANSI C and there is a small portion of code is written in assembly code, and therefore is very easy to port from one platform to another.
- It is provided in form of software components and the system configuration table so that the developers are able to scale the system flexibly according to their requirements.
- Provides support for most of the third-party development tools.

### 3.2 The Real-Time Kernel

The  $\delta$ -Core provides the following functionalities:

- **Initialization Management.** It achieves the initialization of the real-time kernel, and starts the real-time scheduler.
- **Task Management.** It is central to the real-time kernel. It achieves the management of the tasks running on the operating system and provides the functions such as task creation, task deletion, task suspension, task resumption and setting the priori-

# HIGH AVAILABILITY

ty of the tasks.

- Clock Management. It provides support for the application system to respond the external events in real-time in order to make the entire system safely and correctly. This module allows the application programs to set and get system clock; allows the application programs to delay itself in a certain duration or till a specific time; achieves the system timing and the timing of the application programs.
- Interrupt Management. This module achieves the management of interrupt services. It supports
  - The real-time context switching.
  - The stack switching during the nested interruption.
- The Management of Communication and Synchronization. This module provides the supports for the communication and synchronization of tasks. It provides three mechanisms:

- Message boxes for the communication between tasks.
- Semaphores for the resource sharing and mutual exclusion, and the synchronization between the tasks.

## 4 HIGH AVAILABILITY

### 4.1 Reliability

One of the major goal of building a computer system was to make it more reliable.

**Definition 4.1** [Definition of Reliable Systems] A reliable system is defined as a system which will never reach to an error state. In the other words, a reliable system will contain no bugs.

#### 4.1.1 Reliable State

We start from the definition of reliable state.

**Definition 4.2** A reliable state is a state in which its run-

ning task is never going to an error state.

```
def Reliable =
  \ (r : Env, c : Cont, z : State)
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let nz = @(c, z) in @(Neg0, @(Equal, State, nz, ERR_STATE));
```

The definition given above did not eliminate the possibility of the current running task is the idle task. We then have the following rather stronger condition.

```
def Reliable2 =
  \ (r : Env, c : Cont, z : State)
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  @(And,
    @(Not, @(Equal, Identifier, tn, IDLE_TNAME)),
    let nz = @(c, z) in @(Not, @(Equal, State, nz, ERR_STATE)));
```

#### 4.1.2 Weak Reliable System

With the definition of reliable state, we will be able to define the weak reliable systems.

**Definition 4.3** A weak reliable system is a system where there is a state for which its running task is never going to an error state.

In terms of PowerEpsilon, we have:

```
def WeakReliableSys =
  \ (r : Env, c : Cont)
  ? (z : State) [@(Not, @(Equal, State, z, ERR_STATE)) -> @(Reliable, r, c, z)];

def WeakReliableSys2 =
  \ (r : Env, c : Cont)
  ? (z : State)
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  [@(Not, @(Equal, State, z, ERR_STATE)) ->
    @(And,
      @(Not, @(Equal, Identifier, tn, IDLE_TNAME)),
      let nz = @(c, z) in @(Not, @(Equal, State, nz, ERR_STATE)))]];
```

#### 4.1.3 Reliable Systems

A weak reliable system is too weak to be useful. We now give a rather strong definition, -- a reliable system.

**Definition 4.4** A reliable system is a system where for every states its running task is never going to an error state. In terms of PowerEpsilon, we have:

```
def ReliableSys =
  \(\r : Env, c : Cont)
    !(z : State) [@(Not, @(Equal, State, z, ERR_STATE)) -> @(Reliable, r, c, z)];

def ReliableSys2 =
  \(\r : Env, c : Cont)
    !(z : State) [@(Not, @(Equal, State, z, ERR_STATE)) -> @(Reliable2, r, c, z)];
```

#### 4.1.4 Strongly Reliable Systems

We may still give another stronger version of reliable system definition.

**Definition 4.5** A strongly reliable system is a safe and reliable system.

In terms of PowerEpsilon, we have:

```
def StReliableSys =
  \(\r : Env, c : Cont)
    !(z : State)
      [@(Not, @(Equal, State, z, ERR_STATE)) ->
        @(&And1, @(&SafeSystem, r, z), @(Reliable, r, c, z))];

def StReliableSys2 =
  \(\r : Env, c : Cont)
    !(z : State)
      [@(Not, @(Equal, State, z, ERR_STATE)) ->
        @(&And1, @(&SafeSystem, r, z), @(Reliable2, r, c, z))];
```

#### 4.2 High Available Systems

The condition provided above requires that the system has never an error state. This requirement is too strong to be acceptable. We then try to define a rather relaxed condition called high-availability. We will allow that the system goes to an error state but require that it should recover from the wrong state eventually.

High availability is defined as a characteristic of a system that requires as high an MTBF (Mean Time Between Failures) as possible: minimal or zero down time. High-availability systems are required to provide the flexibility to upgrade hardware or software during run-time, or to perform run-time debugging. Hot-swapping, a method of changing or upgrading hardware or software during run-time, is at last becoming a viable option for large and complex systems where shut-downs translate into significant losses in revenue or potential loss of life.

There are several features included:

- Dedicated watch-dog modules;
- Hot-plug connectors;
- Dual-bus architecture;
- Redundant hardware;
- Mirrored operations;
- Dispersion of data.

In  $\delta$ -Core we have provided a simple exception and fatal error handling mechanism by restarting the failed task. This may significantly increase the availability of the system. However, it is hardly to discuss other issues of high-availability without specifying the hardware support.

##### 4.2.1 A Simple Model

We first propose a simple model of HA.

**Definition 4.6** A highly available state is a state when it goes to an error state  $zz$  there will be a subsequent state  $nzz$  of  $zz$  such that  $nzz$  is not an error state.

In terms of PowerEpsilon, we have:

```
def HighAvailable =
  \(\r : Env, c : Cont, z : State)
    let ti = @(GET_CUR_TNAME, z) in
    let tcb = @(GET_TCB, r, z, ti) in
    let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
      tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
    let zz = @(c, z) in
    [@(Equal, State, zz, ERR_STATE) ->
      ?(nzz : State)
        @(&And,
          @(&SameState, zz, nzz),
          let nti = @(GET_CUR_TNAME, nzz) in
          let ntc = @(GET_TCB, r, nzz, nti) in
          let ntn = @(GET_TCB_NAME, ntc), ntm = @(GET_TCB_MODE, ntc),
            ntq = @(GET_TCB_PRIOR, ntc), ntc = @(GET_TCB_CODE, ntc) in
          @(&And,
            @(&Equal, Identifier, ti, nti),
            @(&Not, @(&Equal, State, nzz, ERR_STATE))))];
```

**Definition 4.7** A highly available system is a system where every its states is highly available.

In terms of PowerEpsilon, we have:

# HIGH AVAILABILITY

```
def HASys =
  \ (r : Env, c : Cont)
  !(z : State) [@(Not, @(Equal, State, z, ERR_STATE)) -> @(HighAvailable, r, c, z)];
```

**Definition 4.8** A strong highly available system is a safe and highly available system.

In terms of PowerEpsilon, we have:

```
def StHASys =
  \ (r : Env, c : Cont)
  !(z : State)
  [@(Not, @(Equal, State, z, ERR_STATE)) ->
   @(And, @(SafeSystem, r, z), @(HighAvailable, r, c, z))];
```

## 4.2.2 MTBF Model

When the system recovered from the error state, we usually hope that the recovery is done as soon as possible.

```
def HighAvailable2 =
  \ (r : Env, c : Cont, z : State)
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  [@(Equal, State, zz, ERR_STATE) ->
   ?(nzz : State, q : @(SameState, zz, nzz), nc : Nat)
   let cc = @(FST, q) in
   let nti = @(GET_CUR_TNAME, nzz) in
   let ntcb = @(GET_TCB, r, nzz, nti) in
   let ntn = @(GET_TCB_NAME, ntcb), ntm = @(GET_TCB_MODE, ntcb),
     ntq = @(GET_TCB_PRIOR, ntcb), ntc = @(GET_TCB_CODE, ntcb) in
   let ncc = @(COMPLEX_STATE, cc, r, c, zz) in
   @(And,
    @(NLe, ncc, nc),
    @(And,
     @(Equal, Identifier, ti, nti),
     @(Not, @(Equal, State, nzz, ERR_STATE))))];

def HASys2 =
  \ (r : Env, c : Cont)
  !(z : State) [@(Not, @(Equal, State, z, ERR_STATE)) -> @(HighAvailable2, r, c, z)];

def StHASys2 =
  \ (r : Env, c : Cont)
  !(z : State)
  [@(Not, @(Equal, State, z, ERR_STATE)) ->
   @(And, @(SafeSystem, r, z), @(HighAvailable2, r, c, z))];
```

## 4.2.3 Basic Properties

**Theorem 4.1** A highly reliable state must be highly available. In terms of PowerEpsilon, we have:

```
dec HATHml :
  \ (r : Env, c : Cont, z : State)
  [@(Reliable, r, c, z) -> @(HighAvailable, r, c, z)];
```

The proof is given as follows:

```
def HATHml =
  \ (r : Env, c : Cont, z : State)
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  \ (p : @(Reliable, r, c, z))
  let zz = @(c, z) in
  \ (q : @(Equal, State, zz, ERR_STATE))
  @ (p, q,
   ?(nzz : State)
   @ (And,
    @(SameState, zz, nzz),
    let nti = @(GET_CUR_TNAME, nzz) in
    let ntcb = @(GET_TCB, r, nzz, nti) in
    let ntn = @(GET_TCB_NAME, ntcb),
      ntm = @(GET_TCB_MODE, ntcb),
      ntq = @(GET_TCB_PRIOR, ntcb),
      ntc = @(GET_TCB_CODE, ntcb) in
    @ (And,
     @(Equal, Identifier, ti, nti),
     @(Not, @(Equal, State, nzz, ERR_STATE)))));
```

It is obvious that MTBF is a more restricted model than the simple model. We then have **Theorem 4.2** MTBF model implies the simple model. In terms of PowerEpsilon, we have:

```

dec HATHm2 :
  !(r : Env, c : Cont, z : State)
  [@(HighAvailable2, r, c, z) -> @(HighAvailable, r, c, z)];
The proof is given as follows:

def HATHm2 =
  \ (r : Env, c : Cont, z : State)
  \ (p : @(HighAvailable2, r, c, z))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  \ (u : @(Equal, State, zz, ERR_STATE))
  let w = @(p, u) in
  let nzz = @(FST, w),
    q = @(FST, @(SND, w)),
    nc = @(FST, @(SND, @(SND, w))),
    v = @(SND, @(SND, @(SND, w))) in
  let cc = @(FST, q) in
  let nti = @(GET_CUR_TNAME, nzz) in
  let ntc = @(GET_TCB, r, nzz, nti) in
  let ntn = @(GET_TCB_NAME, ntc),
    ntm = @(GET_TCB_MODE, ntc),
    ntq = @(GET_TCB_PRIOR, ntc),
    ntc = @(GET_TCB_CODE, ntc) in
  let ncc = @(COMPLEX_STATE, cc, r, c, zz) in
  let g = @(PJ2,
    @(NLe, ncc, nc),
    @(And,
      @(Equal, Identifier, ti, nti),
      @(Not, @(Equal, State, nzz, ERR_STATE))),
    v),
    h = @(ANDS,
      @(SameState, zz, nzz),
      @(And,
        @(Equal, Identifier, ti, nti),
        @(Not, @(Equal, State, nzz, ERR_STATE))),
      q,
      g) in
  <nzz, h>;

```

**Theorem 4.3** A highly reliable system must be highly available. In terms of PowerEpsilon, we have:

```

dec HATHm3 : !(r : Env, c : Cont) [@(ReliableSys, r, c) -> @(HASys, r, c)];

```

The proof is given as follows:

```

def HATHm3 =
  \ (r : Env, c : Cont)
  \ (p : @(ReliableSys, r, c))
  \ (z : State, u : @(Not, @(Equal, State, z, ERR_STATE)))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  \ (q : @(Equal, State, zz, ERR_STATE))
  @(p, z, u, q,
    ?(nzz : State)
    @(And,
      @(SameState, zz, nzz),
      let nti = @(GET_CUR_TNAME, nzz) in
      let ntc = @(GET_TCB, r, nzz, nti) in
      let ntn = @(GET_TCB_NAME, ntc),
        ntm = @(GET_TCB_MODE, ntc),
        ntq = @(GET_TCB_PRIOR, ntc),
        ntc = @(GET_TCB_CODE, ntc) in
      @(And,
        @(Equal, Identifier, ti, nti),
        @(Not, @(Equal, State, nzz, ERR_STATE)))));

```

**Theorem 4.4** A reliable system implies a weakly reliable system. In terms of PowerEpsilon, we have:

```

dec HATHm4 :
  !(r : Env, c : Cont) [@(ReliableSys, r, c) -> @(WeakReliableSys, r, c)];

```

# HIGH AVAILABILITY

Assuming that there is an initial state INIT\_STATE for the system, the proof is given as follows:

```
dec INIT_STAT : State;

def HATHm4 =
  \ (r : Env, c : Cont)
  \ (p : @(ReliableSys, r, c)) let z = INIT_STAT in <z, @(p, z)>;
```

**Theorem 4.5** A reliable system implies a MTBF highly available system. In terms of PowerEpsilon, we have:

```
dec HATHm5 :
  ! (r : Env, c : Cont) [@(ReliableSys, r, c) -> @(HASys2, r, c)];
```

The proof is similar to HATHm3.

```
def HATHm5 =
  \ (r : Env, c : Cont)
  \ (p : @(ReliableSys, r, c))
  \ (z : State, u : @(Not, @(Equal, State, z, ERR_STATE)))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
  tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  \ (q : @(Equal, State, zz, ERR_STATE))
  @ (p, z, u, q,
  ? (nzz : State, q : @(SameState, zz, nzz), nc : Nat)
  let cc = @(FST, q) in
  let nti = @(GET_CUR_TNAME, nzz) in
  let ntc = @(GET_TCB, r, nzz, nti) in
  let ntn = @(GET_TCB_NAME, ntc), ntm = @(GET_TCB_MODE, ntc),
  ntq = @(GET_TCB_PRIOR, ntc), ntc = @(GET_TCB_CODE, ntc) in
  let ncc = @(COMPLEX_STATE, cc, r, c, zz) in
  @ (And,
  @ (NLe, ncc, nc),
  @ (And,
  @ (Equal, Identifier, ti, nti),
  @ (Not, @(Equal, State, nzz, ERR_STATE)))));
```

## 4.2.4 Properties for $\delta$ -Core

We now prove that the  $\delta$ -Core is a highly available system.

**Theorem 4.6**  $\delta$ -Core is a highly available system. In terms of PowerEpsilon, we have:

```
dec DeltaHATHm :
  ! (p : Program)
  let c = @(p, Cont, \ (s : Statement) @ (STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  @ (HASys, r, c);
```

When the system went to an error state zz, we will show that @(EXCP\_CONT, r, zz) is subsequent error-free state.

```
dec DeltaHALem :
  ! (p : Program)
  let c = @(p, Cont, \ (s : Statement) @ (STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  ! (z : State)
  [@(Not, @(Equal, State, z, ERR_STATE)) ->
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
  tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  [@(Equal, State, zz, ERR_STATE) ->
  let nzz = @(EXCP_CONT, r, zz) in
  @ (And,
  @ (SameState, zz, nzz),
  let nti = @(GET_CUR_TNAME, nzz) in
  let ntc = @(GET_TCB, r, nzz, nti) in
  let ntn = @(GET_TCB_NAME, ntc),
  ntm = @(GET_TCB_MODE, ntc),
  ntq = @(GET_TCB_PRIOR, ntc),
  ntc = @(GET_TCB_CODE, ntc) in
  @ (And,
  @ (Equal, Identifier, ti, nti),
  @ (Not, @(Equal, State, nzz, ERR_STATE))))];
```

```

def DeltaHATHm =
  \ (p : Program)
  let c = @(p, Cont, \ (s : Statement) @(STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  \ (z : State, w : @(Not, @(Equal, State, z, ERR_STATE)))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  \ (q : @(Equal, State, zz, ERR_STATE))
  let nz = @(EXCP_CONT, r, zz), nq = @(DeltaHALem, p, z, w, q) in
  <nz, nq>;

```

**Theorem 4.7**  $\delta$ -Core is a MTBF highly available system. In terms of PowerEpsilon, we have:

```

dec DeltaHATHm2 :
  !(p : Program)
  let c = @(p, Cont, \ (s : Statement) @(STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  @(HASys2, r, c);

```

If the EXCP\_CONT,  $r$ ) is deterministic with  $nc$  as the upper bound, then the proof of theorem is easily given as follows:

```

dec DeltaHALem21 :
  !(p : Program)
  let c = @(p, Cont, \ (s : Statement) @(STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  !(z : State)
  [@(Not, @(Equal, State, z, ERR_STATE)) ->
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  [@(Equal, State, zz, ERR_STATE) ->
  let nzz = @(EXCP_CONT, r, zz) in @(SameState, zz, nzz)]];

dec DeltaHALem22 :
  !(p : Program)
  let c = @(p, Cont, \ (s : Statement) @(STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  !(z : State, w : @(Not, @(Equal, State, z, ERR_STATE)))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
    tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  !(q : @(Equal, State, zz, ERR_STATE))
  let nzz = @(EXCP_CONT, r, zz),
    nq = @(DeltaHALem21, p, z, w, q),
    nc = @(COMPLEX_STATE, @(EXCP_CONT, r), r, \ (x : State) x, zz) in
  let cc = @(FST, nq) in
  let nti = @(GET_CUR_TNAME, nzz) in
  let ntc = @(GET_TCB, r, nzz, nti) in
  let ntn = @(GET_TCB_NAME, ntc),
    ntm = @(GET_TCB_MODE, ntc),
    ntq = @(GET_TCB_PRIOR, ntc),
    ntc = @(GET_TCB_CODE, ntc) in
  let ncc = @(COMPLEX_STATE, cc, r, c, zz) in
  @(And,
    @(NLe, ncc, nc),
    @(And,
      @(Equal, Identifier, ti, nti),
      @(Not, @(Equal, State, nzz, ERR_STATE))));

```

```
def DeltaHATHm2 =
  \ (p : Program)
  let c = @(p, Cont, \ (s : Statement) @(STAT, s, EMPTY_ENV, @(SCHEDULE, INIT_ENV, IDLE))) in
  let r = INIT_ENV in
  \ (z : State, w : @(Not, @(Equal, State, z, ERR_STATE)))
  let ti = @(GET_CUR_TNAME, z) in
  let tcb = @(GET_TCB, r, z, ti) in
  let tn = @(GET_TCB_NAME, tcb), tm = @(GET_TCB_MODE, tcb),
  tq = @(GET_TCB_PRIOR, tcb), tc = @(GET_TCB_CODE, tcb) in
  let zz = @(c, z) in
  \ (q : @(Equal, State, zz, ERR_STATE))
  let nzz = @(EXCP_CONT, r, zz),
  nq = @(DeltaHALem21, p, z, w, q),
  nc = @(COMPLEX_STATE, @(EXCP_CONT, r), r, \ (x : State) x, zz),
  nw = @(DeltaHALem22, p, z, w, q) in
  <nzz, nq, nc, nw>;
```

## 5 CONCLUSIONS

The main contribution of this work was to use denotational semantics approach for modelling the basic concepts of HA underlying in an operating system. The mathematical properties for HA are investigated ■

---

*Professor Ming-Yuan Zhu was the cofounder and is currently CEO CoreTek Systems. He has spent the last five years on developing the embedded software technology and business for telecommunication, datacommunication, aerospace and consumer electronics industry, and has brought the company to become a leading company in embedded software in China.*

*He had served as the member of program committee of International Computer Software and Application Conference (COMPSAC) from 1992-1995, and is currently the professor Beijing University of Aeronautics and Astronautics and the guest professor of Beijing Institute of Technology.*

*Zhu was born Changchun, China, in 1954 and received a Bachelor degree of Engineering from Changsha Institute of Technology of People's Republic of China in 1982 and a Master of Engineering from the Katholieke Universiteit Leuven, Belgium 1985. Before joining CoreTek Systems, he had been a senior research fellow in Beijing Institute of Systems Engineering and has been in charge of several research projects including.*

*Professor Guang-Ze Xiong was born in 1938. He graduated from Department of Computer Science of Chengdu Institute of Telecommunication Engineering in 1962. His major research areas are real-time operating systems and development tools and techniques for embedded applications. He is now professor and the director of Research Laboratory of Real-Time Software Technology in University of Electronic Science and Technology at Chengdu. He was the cofounder and is now chief engineer of CoreTek Systems.*

*Professor Lei Luo was born in 1963. She graduated and received her master of engineering from Department of Computer Science of University of Electronic Science and Technology at Chengdu in 1987 and 1992. Her major research areas are embedded real-time operating systems and networking techniques. She is now associate professor of Department of Computer Science of University of Electronic Science and Technology at Chengdu. She was the cofounder and is now CTO of CoreTek Systems.*

## REFERENCES

1. R. L. Constable and et al. Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.
2. T. Coquand and G. Huet. The calculus of constructions. Information and Computation, 76(2/3), 1988.
3. M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. LNCS 78. Springer-Verlag, 1979.
4. P. Martin-Löf. Intuitionistic Type Theory. Studies in Proof Theory, Vol. 1. Bibliopolis, Naples, 1984.
5. M.-Y. Zhu. AUTOSTAR -- a software development system. ACM SIGPLAN Notices, 24(3), March 1989.
6. M.-Y. Zhu, Lei Luo, and Guang-Ze Xiong. A provably correct operating system:  $\delta$ -Core. ACM Operating Systems Review, 35(1), 2001.
7. M.-Y. Zhu and C.-W. Wang. A higher-order lambda calculus: PowerEpsilon. Technical report, Beijing Institute of Systems Engineering, Beijing, 1991.
8. M.-Y. Zhu and C.-W. Wang. Program derivation in PowerEpsilon. In Proceedings of COMPSAC'92, Chicago, September 1992.