IEC61508-3 iLE EMNiYETLi YAZILIM GELİŞTİRME

Nadir Subaşı[1], İlker Üstoğlu[2], Mustafa Seçkin Durmuş[3]
[1]Teknik Bilimler Meslek Yüksekokulu, Kırklareli Üniversitesi, Kırklareli, Türkiye
[2]Kontrol ve Otomasyon Mühendisliği Bölümü, Yıldız Teknik Üniversitesi, Davutpaşa Yerleşkesi
[3]Elektrik-Elektronik Mühendisliği Bölümü, Pamukkale Üniversitesi, Kınıklı Yerleşkesi
[1]nadir.subasi@klu.edu.tr, [2]ustoglu@yildiz.edu.tr, [3]msdurmus@pau.edu.tr

ÖZET

Bu çalışmada üçüncü seviye yazılım emniyet bütünlük seviyesine sahip bir yazılım için IEC 61508-3 standardında şiddetle tavsiye edilen yöntemler ele alınmış, çalışmada IEC 61508-3 standardı üzerinde yoğunlaşmış ve SIL3 seviyesindeki bir emniyet yazılımı için isterler konusu üzerinde durulmuştur. Geliştirme ortamı olarak IAR derleyici kullanılmış, ayrıca PC-Lint ve MISRA C++ oluşturulan C++ kodunun verimliliğini kontrol etmede kullanılmıştır. Tekil model entegrasyonu için kod örnekleri verilmiştir. Verilen bu örneklerin, emniyetli yazılım geliştirmeye katkılar sağlaması beklenmektedir.

Anahtar Kelimeler: yazılım emniyet bütünlük seviyesi, MISRA, IAR, PC-Int, IEC 61508-3, modüler yaklaşım, statik kod analizi

SAFE SOFTWARE DEVELOPMENT WITH IEC 61508-3

ABSTRACT

In this study, the highly recommended requirements given in the IEC 61508-3 standard for software with a third level safety integrity level were presented. Especially, the standard, IEC 61508-3, was focused on better developments. SIL3 safety level software's requests have been reviewed. IAR compiler, PC-Lint and MISRA C++ were used for controlling C++ code efficiency. Sample code snippets are given for integration of Singleton Model. Finally, it is expected that the code samples given in this article might provide to safe software and software developments better.

Keywords: software safety, integrity level, MISRA, IAR, PC-Int, IEC 61508-3, modular approach, static code analysis

1. GİRİŞ

Standartlar her nekadar günümüzde geçmişte hiç olmadığı kadar önem kazanmış ve her geçen gün gerek ürünlerin emniyeti gerekse can ve mal güvenliği açısından daha fazla riayet edilir olsalar da genellikle karışık ve yanlış anlaşılmaya müsait metinler içerirler. Havacılık, ulaşım, nükleer ve petrokimya gibi emniyetin kritik olduğu uygulamalarda yazılım geliştirmenin en önemli adımı standartlara uygunluk ve sertifikasyondur.

Standartlarda yer alan gereksinimleri anlamak ve yazılımda bunları gerçeklemek yazılımcının temel görevidir. Gereksinimlerin yanlış anlaşılması ve yanlış uygulanması sistematik arızaların temel nedenlerinden biri olarak gösterilmektedir. Bu problemin bertaraf edilmesi ve standartlara uygun yazılım üretilmesi için çeşitli kalite modelleri önerilmiştir [1]. Bu modeller her bir gereksinim için uygun önlemlerin alınmasını amaçlar ve bu çalışmada odaklanılan C++ dili de dâhil olmak üzere tüm programlama dillerini destekler. Bir yazılım ürününün gözlenmesi ve kontrol edilmesine yönelik olarak yazılım kalitesi ölçütlerinin nasıl ortaya konulacağı ve nasıl kullanılacağı konusu bazı çalışmalarda ele alınmıştır [2], [3].

Bu çalışmada yazılım sertifikasyonu için IEC 61508 fonksiyonel emniyet standardının üçüncü kısmı göz önüne alınmıştır. Bu standart tüm emniyet kritik uygulamalar için şemsiye standart olup, elektrik/elektronik/programlanabilir elektronik emniyet sistemleri için fonksiyonel emniyet sürecini ortaya koyar. Tüm sektörsel emniyet standartları bu standarttan beslenmektedir ki bunlar arasında nükleer alanında IEC61513, demiryolu alanında IEC62279, EN50126, EN50128, EN50129, makine endüstrisinde IEC62061, otomotivde ISO 26262 ve süreç denetiminde IEC 61511 yer almaktadır. Daha önce de

belirtildiği gibi çoğu standartta olduğu gibi bu standartta da bazı yerlerde anlaşılması zor, yorumlanması muhtelif olabilecek metinler yer almaktadır. Birçok standartta yeralan ifadelerin bir tarif vermediği de gözden kaçmayan bir özellik olarak ortaya çıkar. Örneğin olarak bir maddede (Madde 7.4.5.3) yeralan "Yazılım modülerlik, test edilebilirlik ve emniyetli modifikasyon yeteneğine sahip olacak biçimde üretilmelidir." ifadesi yazılım ürününü modülerliği nasıl sağlayacağına dair bir ipucu vermez, somut kurallar ortya koyup kullanıcıya yöntem önermez. Ayrıca yazılımın test edilmesi gerektiği gerçeğini de göz önünde bulundurmak, test ile ilgili zorlukları öngörüp yazılım geliştirme sürecinin ilk fazında bu zorlukların farkında olmak da gerekmektedir. Örneğin son ürünün savunmaya dayalı programlama gibi emniyetli programlama teknikleri kullanarak test edilmesi oldukça zor olabilir. Bu çalışmada IEC 61508-3'te tanımlanmış kurallar ve onların olası kullanımlarına da atıf yapmakta, yazılım emniyet bütünlük seviyesinin (software safety integrity level) SIL3 olması durumu için standardın şiddetle tavsiye ettiği gereksinimler vurgulanmaktadır. Bu çalışmada, IEC 61508-3 [4] gereksinimlerin entegrasyon örnekleri sunulmuştur. İlk olarak geliştiricilerin karşı karşıya kaldıkları sorunlar ele alınmış ve sadece gereksinimler bölümü detaylı olarak incelenmiştir.

2. SIL3 GEREKSİNİMLERİ

SIL 3 seviyesinde bir yazılım emniyet seviyesine ulaşmak için desteklenen araçlar ve programlama dilleri IEC 61508 standardında yer alan bazı teknikler tavsiye edilmektedir [4]. Bu standartları içeren detaylar Tablo 1'de verilmiştir.

Tablo 1. Desteklenen Araç ve Programlama Dilleri

Teknik / Ölçüm	
1	Uygun programlama dili
2	Güçlü yazılmış programlama dili
3	Dil alt kümesi
4a	Sertifikalı araçlar
4b	"Kullanımda kanıtlanmış" araçlar

Aynı standartta yeralan Tablo 1'de ise bu teknikler ve bunlara dayalı programlama dilleri detaylandırılmıştır. Tablo 2'de ise bu detaylandırılmış tekniklerin detayları paylaşmaktadır.

3. GEREKSİNİMLERİN C++ İLE GERÇEKLENMESİ

Emniyet kritik bir uygulamada güçlü yazılmış (strongly typed) bir programlama dili olan C++ ile kod geliştirilecekse emniyet riskleri nedeniyle şablonlar, çoklu miras, reinterpret_cast ve const_cast, istisna işleme (exception handling), mutable olma özelliği ve çalışma zamanı (runtime) tür bilgisi (RTTI) C++ dili işlevleri devre dışı bırakılmalıdır [5]. Öte yandan programlama dilinin düz yazı dosyasından (.cpp ve .h) makine diline çevirmek için sertifikalı araçlar veya kullanımda kanıtlanmış (proven in use) araçlara sahip olması gerekir. Kullanımda kanıtlanmış araçlardan biri IAR'dır [6].

Tablo 2. Detaylı Tasarım

Teknik / Ölçüm	
1a	Yapılandırılmış metotlar
1b	Yarı-biçimsel metotlar
1c	Biçimsel tasarım ve arındırma metotları
2	Bilgisayar destekli tasarım araçları
3	Savunmaya dayalı programlama
4	Modüler yaklaşım
5	Tasarım ve kodlama standartları
6	Yapısal programlama
7	Güvenilir/ doğrulanmış yazılım modülleri ve elemanları kullanma
8	Yazılım emniyet gereksinim şartnamesi ve yazılım tasarımı arasında ileri izlenebilirlik

Tablo 2'de yeralan detaylı tasarım kuralları kaynak kodumuzu nasıl emniyetli kılacağımızı tanımlamaktadır. Örneğin, eğer otomotiv sektöründe emniyet kritik uygulamalar için kod geliştirilecekse MISRA (Motor Industry Software Reliability Association) tarafından C/C++ programlama dili için ortaya konulmuş yazılım geliştirme kurallar kümesine riayet edilmelidir. Bir başka deyişle sadece 61508-3 ile değil aynı zamanda MISRA ile de uyumlu olmak gerekmektedir. MISRA sadece otomotiv sektöründe değil havacılık, haberleşme, tıp elektroniği, savunma, demiryolu başta olmak üzere birçok sektöre kod geliştirenler tarafından da yaygın olarak kabul gören bir model olarak ortaya çıkmıştır [7]. Bu norm ısrarla yazılımın modüler olması gerektiğini belirtir. Modülerlik aynı zamanda kodun kapladığı alanında daha küçük olmasını sağlar. Elbette modüllerin giriş ve çıkışlarının kontrol edilmesi, taşma ve boşalmaların engellenmesi gerekir. Bir modül sertifikalandırılmış ise isterlerin değişmediği durumlarda farklı emniyet uygulamalarında da aynen kullanılabilir. Bazı metotlar aşağıda detaylı incelenmiştir.

A. Yapısal metotlar ve yarı-resmi metotlar

Standart, SIL3 seviye emniyet istendiği bir uygulamada yarı biçimsel metotların kullanılamayacağı durumlarda biçimsel metotların kullanılması gerektiğini belirtir. Yarı biçimsel metotlar C++'da emniyetli yazılım geliştirmek için uygulanabilirdir. Yarı biçimsel metotlar yazılım isterlerinden etkin bir kod elde edebilmek için model veya diyagram kullanmaktadır. UML diyagramları kullanılması durumunda hem isterlerin gerçeklenmesi, hem de emniyet yazılımına dönüştürme işlemi daha kolay olmaktadır

B. Savunmaya dayalı programlama ve modülerlik

Savunmaya dayalı programlama tekniği, yazılım tasarım döngüsünün içerisinde yazılım hatalarını

erken teşhisi kullanılan bir programlama tekniğidir. Şekil 1 ve Şekil 2'de yeralan örneklerde savunmaya dayalı ve dayalı olmayan programlama arasındaki farklar gösterilmektedir.

Şekil 1'de i değişkenine ilk tanımlama yapılmamıştır. Ayrıca while döngüsünde bellekteki olası bir bozulma döngüyü sonsuz döngü haline dönüştürebilir. Şekil 2'de ise aynı kod savunmaya dayalı programlama ilkelerine uyarak yazılmıştır. Bu ilkelerden bazıları, giriş/çıkış işaretleyici ve kaydedicilerin düzenli aralıklarla güncellemesi, aynı girişi birden fazla okuyup ortalamasının alınması, kullanılmayan RAM bellek test etmek, kullanılmayan belleğin bir komut ile doldurulup program sayacının bilinen en son konuma döndürülmesi, sistemin askıda kalmasını önlemek için özellikle sertifikalı harici bir bekçi (watchdog) kullanılması olarak verilebilir. Örnek olarak donanım modüllerinden biri olan tarayıcıyı ele alalım. Tarayıcı ile iletişime geçmek için C++ içerisinde tarayıcı için ilgili modülü yazmanız gerekmektedir.

```
int i;
while(i != 20)
{
cout<<"i değişkeni= "<<i<< endl;i++;
}
```

Şekil 1. Savunmaya dayalı olmayan programlama

```
int i(0);
while(i < 20)
{
cout << "i değişkeni ="<<i<< endl;
i++;
}
```

Şekil 2. Savunmaya dayalı programlama

```
class tarayici { public:
Init_tarayici()
Write_Data();
Read_Data();
...
private:
...
}
```

Şekil 3. Tarayıcı sınıfı

Şekil 3'te sınıfı tanımlanmış olan tarayıcı ile iletişim kurmak için Şekil 4'deki gibi bir kod yazılmalıdır.

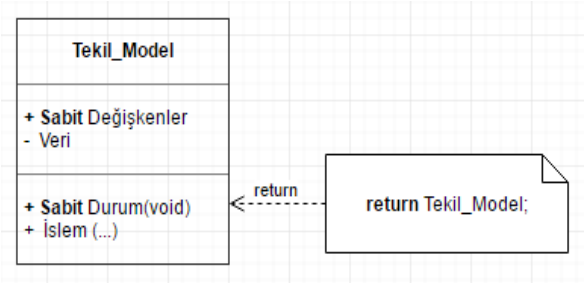
Varsayalım ki tek tarayıcımız olsun ancak biz burada iki tane nesneyi ilklemiş (initialize) olduk öyle ki nesneler eş zamanlı olarak yazma ve okuma yapacak ve neticesinde tarayıcı verilen komutlara doğru cevap vermeyecektir. Bu problemin basit çözümü tarayıcı sınıfının sadece tek bir nesneye sahip olmasıdır. Yarı biçimsel bir yöntem olarak UML diyagramları kullanılarak bir tekil model olarak tanımlanarak bu sorun giderilebilir [8].

```
tarayici              tarayici_A,
tarayici_B,;
tarayici_A. Init_tarayici();
tarayici_B. Init_tarayici();
tarayici_A. Read_Data();
tarayici_B. Read_Data();
```

Şekil 4. Tarayıcı sınıfının kullanımı

```
class tarayici: public Devices
{
public:
static tarayici& Instance(void);
private:
tarayici(void);
tarayici (tarayici const& );
tarayici& operator =(tarayici
const&);
~ tarayici();
}
```

Şekil 5. Tekil Model Sınıfı (UML Diyagram)



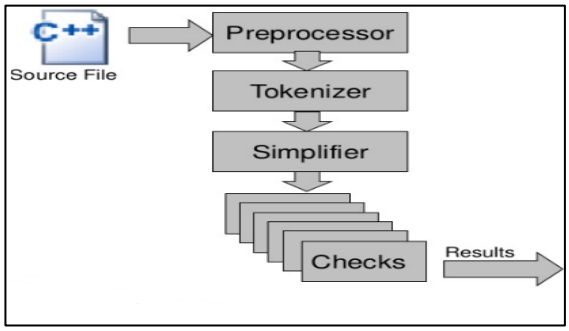
Şekil 6. Tarayıcı Tekil Model Sınıfı

Şekil 6'da verilen çözüm ile kullanıcının birden fazla nesne oluşturulmasına engel olunur. Kullanıcı metotlara erişmesine engel olunarak varsayılan yapıcı metot, kopya yapıcı metot ve kopya atama operatörü özel olarak tanımlanmaktadır. Tarayıcı nesnesini tanımlama için tek yol Instance metodunu kullanmaktır. Instance metodu statik bir metot olup birden fazla nesne oluşturmayı engeller. Ayrıca tarayıcı sınıfı cihazlar sınıfından bir alt sınıftır. Böylelikle yeni cihazlar kayıt edebilir ve silebilir. Kullanılacak programlama dillerinin kendi standartlarına uygun olmasının yanında Emniyet kritik yazılım kodları çalışma alanına uygun standarda göre yazılması gerekmektedir. MISRA bu bağlamda çok fazla sayıda kurallar içermektedir, dolayısı ile standartlara uyumun sınanması için ayrı yazılımlara ihtiyaç duyulur [9].

C/C++ için PC-Lint bu tür yazılımlar arasında yer almaktadır [10]. Tüm statik kod analiz programların genel işleyiş mantığı aşağı yukarı Şekil 7'de gösterildiği gibidir. Böylece koddaki hatalar, kusurlar ve eksiklikler kontrol edilmiş olur. Öte yandan bazı

PC-Lint hata mesajlarının göz ardı edilmesi özelliği de mevcuttur.

C ++ güçlü yazılan bir programlama dili olduğundan sıklıkla karşılaşılan typedef komutunun yanlış kullanımı PC-Lint tarafından tespit edilebilmektedir. Kodun statik olarak analiz edilmesine örnek olarak Şekil 8'de verilen C++ kodu ve Şekil 9'da PC-Lint'in analizi incelenebilir. Ayrıca, araç C++ yerel değişkenlerinin değerlerini izler ve olası sıfır bölünme hatalarını algılar.



Şekil 7. Statik Örnek Kod

```
file.cpp:5 I 1732 new in constructor for class Base which has no assignment operator
file.cpp:5 I 1733 new in constructor for class Base which has no copy constructor
file.cpp:5 I 737 Loss of sign in promotion from int to unsigned int
file.cpp:5 W 1541 Member Base::mp (line 4) possibly not initialized by constructor
file.cpp:7 W 424 Inappropriate deallocation (delete) for 'new[]' data.
file.cpp: 11 W 1511 Member hides non-virtual member Base: :pnntRunType(void) (line 8)
file.cpp:12 W 1569 base class destructor for class 'Base' is not virtual
```

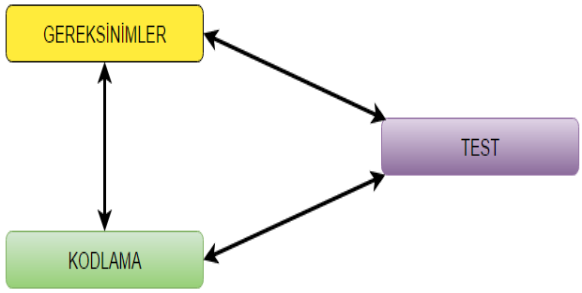
Şekil 8. Örnek C++ Kodu

```
1    class Base {
2        char* mp;
3        int mCount;
4    public:
5        Base(int aCount = 0) { mCount = aCount;
if(mCount > 0) mp = new char(mCount]; }
6        void printRunType() { cout « "Base"; }
7        -Base() { delete mp; }
8    };
9     class Derived : public Base {
10   public:
11       void printRunType() { cout « "Derived"; }
12   };
```

Şekil 9. Örnek C++ Kodu PC-Lint Analiz Sonucu

Nesne tabanlı bir programlama dili olan C++'da iki modülü birbirinden ayırt etmek için sınıflar ve ad alanları kullanılmalıdır. Emniyet yazılımları, sistemde mevcut olan tüm emniyetli donanım ürünlerinin sürücülerine sahip olmalı, tüm dijital sinyaller bir sınıfta ve tüm düşük seviye sürücüler farklı bir ad alanı içerisinde tutulmalıdır. Emniyet yazılımının modüler bir yapısı olması gerekir, her modülün kendine has emniyet fonksiyonu içermelidir. Kullanıcı hangi modülü kullanacaksa bu parçalı yapıdan seçebilir. Ayrıca modüler yaklaşım gelecekte modülün genişletilmesine olanak tanımaktadır.

Standarda göre en önemli gereksinimlerden biri de ileri izlenebilirlik gereksinimidir. Tipik emniyetli yazılım geliştirme döngüsü isterler ile başlamaktadır. İster yazarları ve kod geliştiriciler arasında isterlerin yanlış anlaşılması sorunu oldukça sık rastlanan bir durumdur. Test eden kişi kodu anlık olarak test etmelidir. İleri izlenebilirlik özel yazılım araçları ile sağlanabilir. İsterler değiştiğinde bu durum bir geri bildirim sistemi aracılığı ile hem test edici hem de geliştiriciye bildirilmelidir. Bu prensip Şekil 10'da verilmiştir. İsterleri yazanlar geliştirici ve test edenleri uyarmak zorundadır. Günümüzde bu görevi yazılım araçları yerine getirmektedir.



Şekil 10. İleri İzlenebilirlik

C. Dinamik analiz

Emniyet yazılımı sıkı bir şekilde test edilmeli ve ardından bu testler dokümanlaştırılmalıdır. Test spesifikasyon dokümanı adı verilen bu dokümanlarda test isterleri ve kod entegrasyonu için yol haritası verilmelidir. Burada, C++ programlama dilinin, sonsuz döngüler sınırları aşmak, pointer kullanımı gibi emniyet açısından kritik noktaları dikkate alınmalıdır. Öte yandan yazılım testleri birim testleri ile bitmemektedir, farklı kapsama ölçütleri de kullanılmalıdır. Test edenler verimli kodu değiştirmezler ve geliştiricilere kodu değiştirmelerini söylerler, bu işlemde yazılım geliştirme döngüsünde geride kalındığı için pahalıya mal olur. Emniyet yazılımının test edilebilirliği arttırmak için Geliştirici her özel sınıfın özellikleri için Get ve Set metotlarını tanımlanmalıdır. Diğer taraftan, kapsülleme, bu yöntemlerin kullanımından etkilenmez.

4. SONUÇ (CONCLUSION)

Emniyet yazılım standartlarının entegrasyonu oldukça zordur. Geliştirici genellikle emniyet gereksinimlerini yanlış anlamaktadır. Bundan ötürü çeşitli yazılım

kalite modelleri oluşturmak zorunda kalmışlardır. Bu çalışma, IEC 61508-3 standardı üzerinde yoğunlaşmış ve SIL3 seviyesindeki bir emniyet yazılımı için isterler konusu üzerinde durmuştur. Geliştirme ortamı olarak IAR derleyici kullanılmış, ayrıca PC-Lint ve MISRA C++ oluşturulan C++ kodunun verimliliğini kontrol etmede kullanılmıştır. Tekil model entegrasyonu için basit kod örnekleri verilmiştir.

Bu çalışmada sunulan örneklerin, sunulan yaklaşımın, güvenli yazılım geliştirmeye katkı sağlayacağı, bu konudaki standartların yakınen takip edilerek karşılaşılabilecek tehlikelerin minimize edilebileceği değerlendirilmektedir.

KAYNAKLAR (REFERENCES)

[1] A. Mayr, R. Plösch, M. Saft, Towards an Operational Safety Standard for Software Modelling IEC 61508 Part 3, 18th IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS), 2011, pp. 97 – 104.

[2] I. Sommerville, Software Engineering 9th Edition, Addison-Wesley, 2011.

[3] R. Plösch, H. Gruber, C. Körner, M. Saft, A Method for Continuous Code Quality Management Using Static Analysis, 7th International Conference on the Quality of Information and Communications Technology (QUATIC), 2010, pp. 370 – 375.

[4] International Electrotechnical Commission, IEC, International Standard: 61508 Functional safety of electrical/electronic/ programmable electronic safety-related systems Part 1-7, Geneva, 1999-2010

[5] M. Schreiber, E. Delic, A. Hayek, J. Börcsök, Concept for a SIL3 middleware encapsulating safety-related aspects of applications for an 8051-based SIL3 multi-core system-on-chip, 36th International Convention on Information & Communication Technology Electronics & Microelectronics, 2013, pp. 81-84, ISBN: 978-953-233-076-2.

[6] IAR Systems: https://www.iar.com/.

[7] MISRA: http://www.misra-cpp.com/.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, USA, Addison-Wesley, 1994.

[9] J.L. Anderson, Using software tools and metrics to produce beter quality test software, Proceedings AutoTestCon, pp. 293-297, 2004

[10] J. Gimpel, Software That Checks Software: The Impact of PC-Lint, IEEE Software, vol. 31, no. 1, pp. 15-19, Jan.-Feb. 2014, doi:10.1109/MS.2014.13.