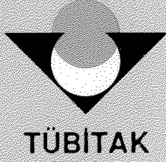
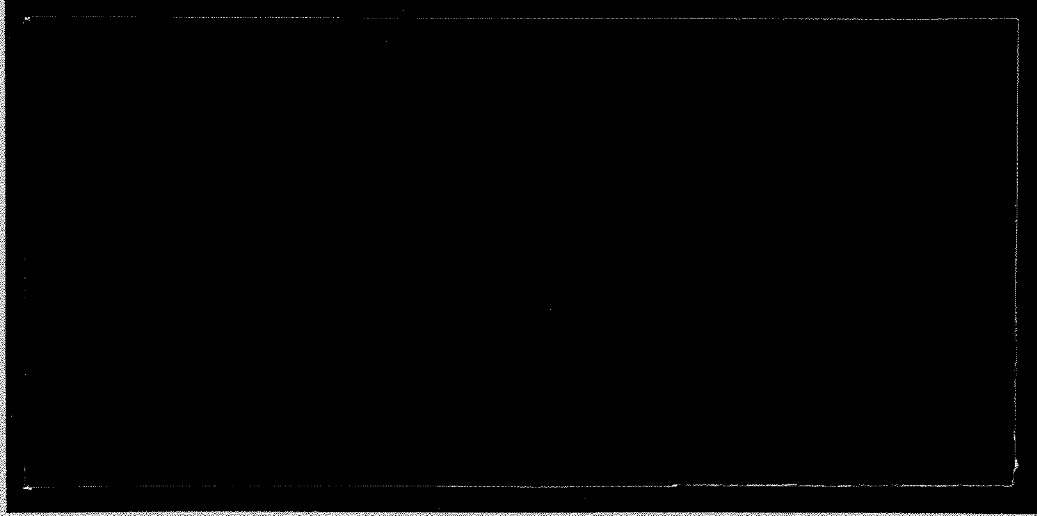


TÜRKİYE BİLİMSEL VE
TEKNİK ARAŞTIRMA KURUMU

THE SCIENTIFIC AND TECHNICAL
RESEARCH COUNCIL OF TURKEY

Dup.

2001/62



Elektrik, Elektronik ve Enformatik Araştırma Grubu

Electric, Electronics and Informatics Research
Grant Committee

TÜRKİYE BİLİMSEL VE
TEKNİK ARAŞTIRMA KURUMU

THE SCIENTIFIC AND TECHNICAL
RESEARCH COUNCIL OF TURKEY

Biçimsel Yöntemler ile Dağılımlı Yazılımların Doğrulanması ve Geçerlileştirilmesi

Proje No: 194E017 (COST-247)

Elektrik, Elektronik ve Enformatik
Araştırma Grubu

Electric, Electronics and Informatics
Research Grant Committee

ÖNSÖZ

Bu proje Avrupa Topluluğu kuruluşu olan COST'a sunulup kabul ettirilmiş olan ve 18 Avrupa ülkesinin katılımı ile başkanlığı ve sekreteryasının Türkiye tarafından yürütüldüğü COST 247 kodlu ve *Verification/Validation of Communication Software by Formal Methods* adlı projenin Türkiye içindeki etkinliklerini TÜBİTAK-EEEAG desteği ile sürdüren projedir.

Projede endüstriyel uygulamaya yönelik olarak SDL dili üzerinde betimlemeler, SDL/COSPAN derleyicisi gibi yazılım aygıtları, entegre donanım/yazılım tasarımı için tanımlar ve aygıtlar ve doğrulayıcılar üzerinde karmaşıklık azaltan (*complexity relief*) teknikler geliştirilmiş ve bunun yanısıra uyum sınama (*conformance testing*) ve ayrık olay dizgeleri (*discrete event systems*) üzerinde kuramsal çalışmalar yapılmıştır.

İÇİNDEKİLER

ÖZET (ABSTRACT) 5

1 – GİRİŞ 6

1.1 – AVRUPA TOPLULUĞU ÇERÇEVESİNDE COST 247 PROJE PLATFORMUNUN OLUŞUMU VE ETKİNLİKLER 6

1.2 – TÜBİTAK COST 247 PROJESİNİN AMACI VE PLATFORMA KATILIM STRATEJİSİ 8

2 – ARAŞTIRMANIN GELİŞME AŞAMALARI 10

2.1 – SDL DİLİ ÜZERİNE ÇALIŞMALAR 10

2.1.2 – SİSTEM BETİMLEME 10

2.1.2 – SİSTEM DOĞRULAMA 10

2.2 – UYUM SINAMA 12

2.3 – YAN CALIŞMA : AYRIK OLAY DİZGE YAKLAŞIMI 14

3 – YAYINLAR VE TEZLER 16

3.1 – YAYINLAR 16

3.2 – TEZLER 17

4 – MALİ HARCAMALAR 19

5 – SONUÇ 20

6 – EKLERİN LİSTESİ 22

BİBLİYOGRAFİK BİLGİ FORMU

EKLER

Şekil 1 – Protokol katmanları arasında bağlantı 15

Tablo 1 – Mali Harcamalar 19

ÖZET

Gerçek zaman iletişim yazılımlarında kullanılan dağımlı yazılımların betimlenmesi ve doğrulanması için iki ticari SDL ürünü olan SDT (Telelogic) ve GEODE (Verilog) paketleri kullanılarak uygulamalar yapıldı ve yeni teknikler geliştirildi. Özellikle Lucent Technologies'de geliştirilmiş olan COSPAN doğrulayıcısının SDL platformunda kullanılabilmesi için SDL-S/R derleyicisi gerçekleştirildi ve SDL dili ile COSPAN doğrulayıcısının donanım/yazılım entegre tasarımı (HW/SW co-design) için genişletilmiş SDL tanımı yapıldı. Bunun yanısıra SDL kullanımında durum patlaması olarak anılan karmaşıklık sorununa çözüm getirmek üzere teknikler geliştirildi.

ABSTRACT

In order to specify and validate embedded real-time software used in communication systems the commercial products SDT and GEODE were applied to numerous examples and techniques were developed for that purpose. In particular to use COSPAN verifier developed at Lucent Technologies within an SDL platform a restricted SDL to S/R compiler was implemented and SDL was extended as the first step of implementing a HW/SW codesign tool using the COSPAN verifier. In addition complexity relief techniques were developed to overcome the problem of state explosion when using SDL

1 – GİRİŞ

1.1 – AVRUPA TOPLULUĞU ÇERÇEVESİNDE COST 247 PROJE PLATFORMUNUN OLUŞUMU VE ETKİNLİKLERİ

COST 247 projesinin bir Avrupa Topluluğu COST projesi olarak oluşumunun ilk tohumları 10-13 Aralık 1992 tarihinde Ankara'da yapılan bir ön çalıştay (*preliminary workshop*)ile atıldı. Bu çalıştayda 9 Avrupa ülkesinden katılan temsilciler kendi istek ve araştırma ilgi alanlarını ortaya koyarak COST Telekomünikasyon grubu içinde yer alacak ve telekom yazılımlarına ilişkin sorunları içerecek bir projeyi yönlendirdiler (bkz. Ek 1). Bu aşamadan sonra COST kuruluşuna yaptığımız proje başvurusu (bkz. Ek 2) COST Telekom grubu tarafından kabul edildi ve katılım kararları ülkelerin imzalarına açıldı. Kurucu toplantı (*inaugural meeting*) 9 Aralık 1993 tarihinde Brüksel'de gerçekleşti. Ancak COST kuruluşunun bir gereği olarak projenin resmen başlaması için en az 5 COST üyesi ülkenin büyükelçiliklerinin imzalarının daha yer almamış olması nedeniyle resmi başlangıç 14,15 Mart 1994 tarihinde yapılan Utrecht yönetim kurulu toplantısında gerçekleşti.
Bu toplantıda, katılan ülke temsilcilerinden oluşan COST 247 yönetim kurulu, öneriyi yapan ülkenin temsilcisi K. İnan'ı proje başkanlığına ve Fransa temsilcisi S. Budkowski'yi ikinci başkanlığa seçti. Aynı toplantıda 4 ayrı çalışma grubu (ÇG) ve başkanları :

(1) E-LOTOS grubu (H. Garavel, Fransa),

(2) Biçimsel betimleme analiz ve doğrulama grubu (J. F. Groote, Holanda),

(3) Uyum sınaması grubu (K. Tarnay, Macaristan),

(4) Simülasyon ve başarım analizi grubu (P. Dembinski, Polonya),

olarak belirlendi. Üç yıllık bir proje olan ve 18 Avrupa ülkesinin katıldığı COST 247 projesi 1996 Aralık ayında sona erdi ve üye ülkelerin isteği ve COST Telekom grubunun onayı ile Aralık 1997 tarihine kadar uzatıldı. Bu tarihte COST kuruluşuna yapılan tüm sunuş özetleri ve diğer etkinlikleri içeren dokümanlarla beraber bir sonuç raporu olarak COST'a sunuldu (bkz. Ek 3) . Bu süre içinde hemen hepsi iki gün süren ve yarım günü yönetim kurul toplantısı ve kalan sürenin teknik sunuşları içerdiği 11 toplantı yapıldı. Bu toplantılara ek olarak 5 tane özel çalıştay toplantısı, bu çalıştay toplantılarından iki tanesinin *proceeding* 'lerini ve *Science of Computer Programming , vol. 29, No. 1 & 2* sayısının "Methods of Software Design : Techniques and Applications" teması altında özel COST 247 sayısını içeren yayın etkinlilerini ve COST tarafından finanse edilen karşılıklı kısa süreli ziyaretleri (*short term missions*) ve E-LOTOS grubunun özel bir çıktısı olan LOTOS dilinin teknolojik gelişmelere uyumlu bir sonraki versiyonunun ITU standardlarını COST 247 platformunun çıktıları arasında sayabiliriz.

COST 247 projesi COST Telekom grubu içinde telekom yazılımları (*embedded real-time software*) son derece kritik bir alanı kapsayan önemli bir boşluğu dolduruyordu. İlgili alanının genişliği, yazılıma ilişkin konuların ticari boyutunun getirdiği özellikler - yani ticari değere sahip bir yazılım ürünü geliştirmenin gerektirdiği bütçenin, COST 247 bütçesinin çok üstünde olması - COST 247 projesini ortaklaşa yürütülen tek amaçlı bir proje yerine, çok amaçlı bir bilgi alışverişi platformu olarak gelişmesini gerektirmiştir. EEEAG COST 247 projesi de bu genel platforma katılan ve daha önce desteklenen EEEAG Yazılım 2 projesinin birikimleri üzerine sürdürülen bir proje olarak gelişti.

1.2 - TÜBİTAK COST 247 PROJESİNİN AMACI VE PLATFORMA KATILIM STRATEJİSİ

EEEAG COST 247 projesi Türkiye'nin kendi önermiş olduğu COST 247 projesine katılımını sağladı. Bu projeyi, gerek birikim, gerek oluşmuş olan altyapı açısından COST 247 ile yaklaşık 9 aylık bir çakışma süresi olan ve onun doğmasını sağlayan EEEAG Yazılım 2 projesinin devamı olarak nitelemek gerekir. Türkiye'deki telekomünikasyon sektörünün görece gelişmişliği belirleyici bir veri olarak alındı ve buna bağlı olarak yaklaşım stratejimizde birkaç istisna dışında akademik yön yerine, pratik yöne daha fazla ağırlık verildi. Örneğin pratik kullanım boyutu gitgide kaybolan LOTOS ve ESTELLE gibi akademik dünyanın doğurduğu dillere ilişkin çalışmalara yer

verilmedi. Bunun yerine ticari uygulaması gitgide yaygınlaşmakta olan SDL diline yönelik çalışmalar ön plana alındı. SDL dilinin ticari kullanım ürünü olarak pazarlanan İsveç kökenli Telelogic şirketinin SDL ürünü olan *SDT* paketi ile Fransa kökenli Verilog şirketinin SDL ürünü olan *GEODE* paketi çalışma boyunca kullanıldı ve COST 247 platformu içinde bu ürünleri kullanan gruplar ile yakın ilişki kuruldu. EEEAG Yazılım 2 projesi içinde satın alınmış olan SDT paketinin bir sonraki versiyonu olan nesneye yönelik ve SDL 92 dilini içeren yeni *SDT* paketinin *GEODE* ile beraber ücretsiz olarak sağlanabilmesinde COST 247 platformu içindeki liderlik konumumuz önemli rol oynadı. Bunun yanısıra, EEEAG Yazılım 2 projesinden beri incelemekte ve kullanmakta olduğumuz Bell-Labs'de geliştirilen *COSPAN* doğrulayıcı aygıtının *FormalCheck* adlı bir donanım doğrulayıcısı olarak piyasaya sürülmesi de, pratik uygulamalar konusundaki öngörülerimizi doğrular nitelikte oldu. Proje boyunca karşılaştığımız en büyük zorluk, genelde yazılım sanayilerinin, mütevazi araştırma kaynaklarını aşan ticari boyutları ve bu boyutların gerektirdiği insan ve finansman kaynaklarına erişememizden doğdu. Başta TÜBİTAK Doprog programı ile getirmiş olduğumuz ve daha sonra NATO desteği, sanayi desteği (TELETAŞ projesi) ve TÜBİTAK-BİLTEN desteği ile projede çalışan Rus bilgisayarcıları V. Levin ve E. Bouminova'nın Bell-labs tarafından kendi ticari atılımlarında (*FormalCheck*) kullanılmak üzere işe alınması, bu görüşümüze somut bir örnek oluşturmakta.

2 – ARAŞTIRMANIN GELİŞME AŞAMALARI

2.1 - SDL ÜZERİNE ÇALIŞMALAR

2.1.1 - SİSTEM BETİMLEME

SDL dilinin dünya piyasasında iki ticari uygulama aygıtı olan *Telelogic SDT* ve *Verilog GEODE* çeşitli uygulamalarda kullanılmıştır. Sistematik bir uygulama EEEAG Yazılım 2 projesi ile çakışan süre içinde gerçekleştirilen TELETAŞ -ALCATEL üretimi Levent Santralının seçilmiş bazı işlevlerinin SDL ile betimlenmesidir (bkz Ek 4). Bunun yanısıra yüksek lisans tezlerinden bir bölümü (bkz. Ek 12 , tezler : 3,7 ve 8) dolaysız olarak, bazıları da (bkz. Ek 12 , tezler : 1,2 ve 9) dolaylı olarak kapsamlı SDL betimlemelerini içermektedir.

2.1.2 - SİSTEM DOĞRULAMA

Projedeki önemli bir etkinlik önce SDL - C++ derleyicisi (animatör) çerçevesinde yapılan ön çalışmalardan sonra (bkz Ek 5 ve Ek 6) *SDT* ve *GEODE* gibi ticari ürünlerin varlığında bunun verimli bir yaklaşım olmayacağı düşünülmüş ve proje, SDL-COSPAN derleyici tasarımına dönüştürülmüştür. Bu tasarımdan önce daha önceki çalışmadan yansıyan kısım SDL'den herhangi bir dile derleme yapılmasında kullanılan ön yüz (*front end*) gösteriminin tasarımı ve gerçekleştirimi ile (bkz. Ek 12 tez 6) genel derleyici arayüz tasarım yaklaşımına ilişkin bir yayın yer almaktadır (bkz. Ek 8). Buradaki yaklaşımın altında yatan

motivasyon, betimleme amacı ile kullanımı görece kolay ve yaygın, ancak semantik yapısındaki boşluklardan dolayı doğrulanması zor olan SDL aygıtlarının, tam tersi özelliklere sahip COSPAN ile dengelenmesiydi. Gerçekten de gerek *SDT* gerek *GEODE*'un doğrulayıcıları denenerek çok yetersiz bulunmuş ve temas ettiğimiz ve bu aygıtları kullanan sanayiler bu veya başka nedenlerden dolayı aygıtın doğrulayıcısının özelliklerinden pek haberdar olmadıklarını ifade etmişlerdir. Öte yandan daha çok donanım doğrulanması amacı ile geliştirilmiş olan COSPAN'in bir yazılım betimlenme aygıtı olarak kullanılmasının çok kısıtlı olacağı, ancak doğrulama özellikleri son derece güçlü olması nedeni ile SDL-COSPAN derleyicisi ile SDL'e nitelikli bir doğrulayıcı sağlanmış olacağı düşünülmüştür. Söz konusu derleyicide karşılaşılan temel zorluk geniş semantikli SDL'in dar semantikli COSPAN'a (senkron iletişimli sonlu durum makinaları) sığdırılması için gereken varsayım ve kısıtların oluşturulmasıydı (ön sonuçlar için bkz. Ek 7, kullanılan derleyici tasarımı için bkz. Ek 8). SDL-COSPAN derleyicisinin kullanılabileceği diğer bir alan, beraber yazılım/donanım tasarımında (*SW/HW co-design*) yazılım ekibinin sistem yazılımını SDL ile ; donanım ekibinin ise sistem donanımını COSPAN ile betimlemesi ve yazılım-donanım arabirimleri için SDL diline ek tanımlar getirerek sonuçta tanımlanan tüm sistemi COSPAN'a diline derlemek ve böylece doğrulamayı COSPAN üzerinden yapabilmek olarak tasarımlandı (bkz. Ek 9). Bu tasarımın bir ürüne dönüşmesi aşamasına ülke

içinde bu konuda herhangi bir talep oluşturacak ve ticari boyutlarda destek verecek bir grup olmadığı için varılamadı.

Son olarak yazında kısmi sıra (*partial order*) teknikleri olarak geçen ve sonlu durumlu makinalarda doğrulamak için erişilmesi gereken durumlarda önemli azaltma sağlamak fikrine dayalı teknikler SDL diline uygulanmış ve doğrulayıcıda sistem karmaşıklığını azaltan kolaylık sağlama teknikleri geliştirilmiş ve uygulanmıştır (bkz. Ek 10 ve Ek 12, A. Şen MS Tezi özeti).

2.2 - UYUM SINAMA

Telekom yazılımları için uyum sınaması konusu sanayide karşılaşılan önemli konulardan biridir. Uyum sınama, belli bir donanım üzerinde yer alan bir yazılımın girdi-çıktı davranış biçiminin verilmiş olan standard davranış modeline uyumlu olmasının sınanmasıdır. Bu konuda yazında yer alan temel kuramsal sonuçlar, sonlu bir girdi-çıktı otomatı modeline uyumu sınamak için sisteme uygulanacak girdi dizininin (denetleme dizini (*checking sequence*)) hesaplanma ve uygulanma yöntemlerini içermektedir. Uyum modelinin durum sayısının çok olması, hatta birçok durumda sonsuz olması sözkonusu denetleme dizininin hesaplanmasını ve/veya uygulanmasını ya pratik-dışı ya da kuramsal olarak olanaksız kılabilmektedir. Bu durumlarda izlenen yöntem çoğu kez somut pratik senaryolarda oluşabilecek girdi dizinlerinin bir sınama dizini olarak sisteme uygulanması

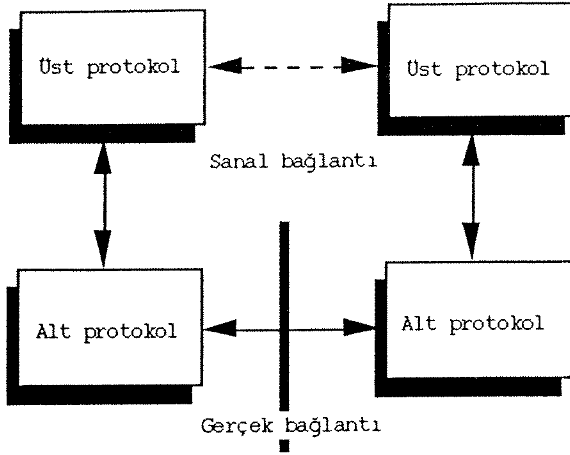
ve izlenen çıktı dizinlerinin beklenen çıktıklar ile karşılaştırılmasıdır. Bu yöntemin en önemli yetersizliği çoğu kez yakalanması zor sistem hatalarının tipik senaryolarda değil hiç beklenmeyen senaryolarda ortaya çıkmasıdır. Bu tür marjinal senaryolar ise sayıca çok olabildiği için hata yakalama veriminin artması genelde konuya daha sistematik bir yaklaşımı gerektirmektedir. Uyum sınama konusunun pratiğe dönük diğer bir boyutu gerçek zaman içinde testlerin uygulanması için gereken test uygulama mimarilerinin özelliklerinden kaynaklanmaktadır. Özellikle sınanacak sistem cografi olarak dağılımlı bir yapıdaysa sınama mimarisinin de dağımlı olma gereği ortaya çıkmakta ve beraber çalışacak sınama birimleri arasında eşgüdüm gerekmektedir. Pratikte kullanılmak üzere sınama dizinleri ve yöntemleri için geliştirilmiş bir ITU standardı olan TTCN *(Tree and Tabular Combined Notation)* sınama dizinlerinin sınıflandırılması, betimlenmesi ve sonuçların yorumlanması üzerinde bir notasyon (formel dil) sistemidir. Uygun test dizinlerinin tanımlanması ve sonuçların bir rapor olarak kaydedilmesi TTCN aygıtları sayesinde (örneğin ticari bir ürün olan SDT aygıtının ek bir TTCN aygıtı mevcuttur) kolaylaştırılmaktadır.

Bu proje çerçevesinde uyum sınama konusundaki yaklaşımımız, diğer yönlerdeki yaklaşımımızın tersine,yani kuramsal yönde olmuştur. İki tane MS tezi (bkz. Ek 12, H. Çalgın ve O. Gücün tez özetleri) çeşitli varsayımlar altında optimal denetleme dizinlerinin hesabını içermektedir. Daha sonra Ottawa

Üniversitesinden H. Ural ile beraber bu sonuçlar, yazında dağınık birçok sonucu ve yenilerini içeren bir makale olarak sunulmuş ve basılmıştır (bkz. Ek 11).

2.3 - YAN CALIŞMA : AYRIK OLAY DİZGE YAKLAŞIMI

Kontrol ve bilgisayar alanlarının kesişmesinde yer alan ayrık olay dizgeleri disiplini ile incelenen bir konu hiyerarşik danışmalı denetim konusuna ilişkindir. Konu aşağıdaki şekilde görüldüğü gibi alt düzeyde çalışan ve gerçek bağlantıyı sağlayan protokolun üst sisteme yansıtılma sorunudur. Alt düzeydeki protokolun verdiği hizmeti üst düzeyde hangi primitiflerin (ayrık olayların) kontrol edilebildiği ve hangilerinin gözlenebildiğine yansıtılması ve böylece alt düzeyde karmaşıklığı ortadan kaldırarak üst düzeyde yazılabilecek yazılımların otomatik olarak alt düzeydeki gereken programlara derlenebilmesini sağlayacak bir derleme düzeni kurmaktır. Bu konudaki çalışma A. Nergis'in doktora çalışması olarak sonuçlandırılmıştır (bkz. Ek 8; A. Nergis tez özeti)



Şekil 1 – Protokol katmanları arasındaki bağlantı

3 – YAYINLAR VE TEZLER

3.1 - YAYINLAR

1 - V. Levin, A. Sulimov, H. Yenigün and K. İnan, "Preliminary Concepts for a C++ Animator Design for SDL 92", published in Russian Academy of Sciences, Novosibirsk Informatics Institute Journal

2 - V. Levin, H. Yenigün and K. İnan,
"An Implementation of SDL 92 Communication Semantics", Proceedings ISCIS X, Oct.30 - Nov. 1, 1995, Kuşadası, Turkey, pp. 457-464.

3 - O. Başbuğoğlu, K. İnan,
"Compiling SDL into the Finite State Specification Language Cospan", Proceedings ISCIS X, Oct.30 - Nov. 1, 1995, Kuşadası, Turkey, pp. 643-650

4 - V. Levin, E. Bounimova and K. İnan,
"An Interface Implementation Using a Compiler Specification Method", Proceedings ISCIS XI, Nov.6-8, 1996, Antalya, Turkey, pp.385-394

5 - V. Levin, E. Bounimova, O. Başbuğoğlu and K. İnan,
"A Verifiable Software/Hardware Co-design Using SDL and COSPAN", Proceedings of COST 247 International Workshop on Applied Formal Methods in System Design, June 17-19,1996 Maribor, Slovenia, pp.6-16.

6 - H. Yenigün, K. İnan,
"A Verifiable Software/Hardware Co-design Using SDL and COSPAN", Proceedings of the Second COST 247 International

Workshop on Applied Formal Methods in System Design, June 18-19,1996 Zagrep, Croatia, pp. 11-16.

7 - K. İnan, H. Ural, "Efficient Checking Sequences for Testing Finite State Machines", Journal of Information and Software Technology, vol. 41 1999, pp. 799-812.

3.2 - TEZLER

1- "SDL ve ERPAL dillerini kullanarak yapılan süreç belirtmelerinin karşılaştırılması", Ali Sezgin, 1994 ODTÜ Y.L. Tezi

2- "Yeni bir cebirsel dil ERPAL ve SDL-92'nin bir altkümesinin ERPAL'e çevrilmesi" , Hüsnü Yenigün, 1995 ODTÜ. Y.L. tezi.

3- "Bir abone kimlik birimi yöneticisinin betimleme dili SDL kullanarak gerçekleştirilmesi" , Cenk Tuğçetin, 1995 ODTÜ. Y.L. tezi.

4- "Ayırt edici diziler ve uyarlanmış kırsal postacı algoritması kullanarak eniyi uyum sınama dizisi üreten bir yöntemin gerçeklenmesi" , Hüsyin Çalgın, 1996 ODTÜ. Y.L. tezi.

5- "Bir sonlu durum makinası için tanımlayıcı kümeler kullanarak test dizisi üretimi" , Onur Gücün, 1996 ODTÜ. Y.L. tezi.

6- "SDL 92 derleyicisi ön-yüzünün tasarımı ve geliştirilmesi" , Uygar Doyuran, 1996 ODTÜ. Y.L. tezi.

7- "ISDN katmanının SDL 92 kullanılarak belirtimi ve geçerlilik sınaması" , Doğuş Çenberci, 1997 ODTÜ. Y.L. tezi.

8- "XPRESS taşıyıcı protokolunun SDL 92 kullanılarak belirtimi ve geçerlilik sınaması" , İsmail Dalkıran, 1997 ODTÜ. Y.L. tezi.

9- "SDL sistemlerinin kısmi sıralama metodları ile doğrulanması" , Alper Şen, 1997 ODTÜ. Y.L. tezi.

10- "Ayrık olay dizgeleri için kontrol içerme/seçme yaklaşımı ve katmanlı kontrole uygulanması" , Aydın Nergis, 1997 ODTÜ. Doktora tezi.

11- "Uyum sınaması üretilmesi için algoritmalar" , Burak Serdar, 1998 ODTÜ. Y.L. tezi.

COST-247	PERSONEL	TEÇHİZAT	SARF	SEYAHAT	HİZMET	BİLGİ İŞLEM	KIRTASİYE	DİĞER	TELİF	TOPLAM	AÇIKLAMA
ÖDENEK94	**300.000.000**	**700.000.000**	**50.000.000**	**300.000.000**	**20.000.000**		**40.000.000**	**40.000.000**		**1.450.000.000**	
AKTARMA	103.000.000			-103.000.000						0	
R.ÖDENEK94	**403.000.000**	**700.000.000**	**50.000.000**	**197.000.000**	**20.000.000**	**0**	**40.000.000**	**40.000.000**	**0**	**1.450.000.000**	
HARCAMA94	**291.044.090**	**0**	**410.000**	**143.851.663**	**2.104.500**	**0**	**1.000.000**	**24.716.000**	**0**	**463.126.253**	
KALAN94	111.955.910	700.000.000	49.590.000	53.148.337	17.895.500	0	39.000.000	15.284.000	0	986.873.747	
ÖDENEK95	**720.000.000**		**50.000.000**	**300.000.000**	**35.000.000**		**30.000.000**	**70.000.000**	**200.000.000**	**1.405.000.000**	
A.ÖDENEK95	**831.955.910**	**700.000.000**	**99.590.000**	**353.148.337**	**52.895.500**	**0**	**69.000.000**	**85.284.000**	**200.000.000**	**2.391.873.747**	
AKTARMA		-55.000.000			55.000.000					0	
R.ÖDENEK95	**831.955.910**	**645.000.000**	**99.590.000**	**353.148.337**	**107.895.500**	**0**	**69.000.000**	**85.284.000**	**200.000.000**	**2.391.873.747**	
HARCAMA95	**385.614.774**	**574.627.500**	**58.176.650**	**256.952.866**	**40.000.000**	**0**	**51.825.902**	**64.266.500**	**39.200.000**	**1.470.664.192**	
KALAN95	446.341.136	70.372.500	41.413.350	96.195.471	67.895.500	0	17.174.098	21.017.500	160.800.000	921.209.555	
ÖDENEK96	**864.000.000**		**50.000.000**	**400.000.000**	**45.000.000**		**30.000.000**	**90.000.000**	**325.000.000**	**1.804.000.000**	
A.ÖDENEK96	**1.310.341.136**	**70.372.500**	**91.413.350**	**496.195.471**	**112.895.500**	**0**	**47.174.098**	**111.017.500**	**485.800.000**	**2.725.209.555**	
AKTARMA		-70.372.000	70.372.000							0	12.9.96
AKTARMA			200.000.000						-200.000.000	0	11.11.96
R.ÖDENEK96	**1.310.341.136**	**500**	**361.785.350**	**496.195.471**	**112.895.500**	**0**	**47.174.098**	**111.017.500**	**285.800.000**	**2.725.209.555**	
HARCAMA96	**1.052.820.777**	**0**	**228.794.736**	**232.780.642**	**5.750.000**	**0**	**47.130.546**	**109.512.000**	**8.400.000**	**1.685.188.701**	
KALAN96	257.520.359	500	132.990.614	263.414.829	107.145.500	0	43.552	1.505.500	277.400.000	1.040.020.854	
ÖDENEK97	**216.000.000**								**225.000.000**	**441.000.000**	
A.ÖDENEK97	**473.520.359**	**500**	**132.990.614**	**263.414.829**	**107.145.500**	**0**	**43.552**	**1.505.500**	**502.400.000**	**1.481.020.854**	
AKTARMA			100.000.000						-100.000.000	0	28.4.1997
AKTARMA		402.400.000							-402.400.000	0	13.5.1997
EK ÖDENEK				250.000.000						250.000.000	
R.ÖDENEK97	**473.520.359**	**402.400.500**	**232.990.614**	**513.414.829**	**107.145.500**	**0**	**43.552**	**1.505.500**	**0**	**1.731.020.854**	
HARCAMA97	**444.975.052**	**270.412.725**	**233.362.588**	**490.217.448**	**0**	**0**	**0**	**0**	**0**	**1.438.967.813**	
KALAN97	28.545.307	131.987.775	-371.974	23.197.381	107.145.500	0	43.552	1.505.500	0	292.053.041	
T.ÖDENEK	**2.203.000.000**	**977.028.000**	**520.372.000**	**1.147.000.000**	**155.000.000**	**0**	**100.000.000**	**200.000.000**	**47.600.000**	**5.350.000.000**	HARCAMA ORANI
T.HARCAMA	**2.174.454.693**	**845.040.225**	**520.743.974**	**1.123.802.619**	**47.854.500**	**0**	**99.956.448**	**198.494.500**	**47.600.000**	**5.057.946.959**	**0,95**
T.KALAN	**28.545.307**	**131.987.775**	**-371.974**	**23.197.381**	**107.145.500**	**0**	**43.552**	**1.505.500**	**0**	**292.053.041**	

5 – SONUÇ

EEEAG COST 247 TÜBİTAK projesi, tarafımızdan önerilip sürdürülmüş olan genel COST 247 projesine Türkiye'nin katılımını finanse etmiştir. Bu proje çerçevesinde çeşitli Avrupa ülkelerinde biçimsel yöntemleri telekom yazılımlarının tasarımında kullanan gruplar ile ilişki kurulmuş ve bu alanda geniş bir deneyim kazanılmıştır. Proje süresi içinde üretilmiş olan makale, bildiri ve diğer yayınların yanısıra üretilen en önemli yazılım ürünü SDL-COSPAN derleyicisi olmuştur. Bu yazılımın üretilmesindeki kazanılan deneyim, bu konuda Türkiye'de bir ticari girişim olmadığı için *Lucent Technologies (Bell-labs)* içinde süren ticari boyutlu çalışmalara, gerek ürün, gerekse yetişmiş eleman (V. Levin, E. Bounimova ve H. Yenigün) olarak katkıda bulunmuştur.

Projenin geleceğe yönelik olarak yapacağı katkılardan bazıları gerçekleşmiş sayılabilir. Örneğin doğrulama sorununda karmaşıklığı aşmak için geliştirilmiş olan yöntem endüstriyek uygulamada kullanılabilecek boyutta ayrıntılandırılmış ve bugün Lucent tarafından ticarileşmiş olan donanım doğrulayıcısı ***FormalCheck*** aygıtına entegre olmaya hazır duruma getirilmiştir. Bu çalışma başından sonuna kadar projede çalışmış olan H. Yenigün'ün Murray Hill, Lucent Technologies'de sürdürdüğü çalışmanın ürünü olup aynı zamanda da ODTÜ'de tamamlamış olduğu doktora tezinin çekirdeğini oluşturmaktadır.

Aynı şekilde projede başlatılmış olan ve SDL-COSPAN bileşik aygıtının HW/SW entegre tasarımı olarak son hale getirilmesi projede yer alan V. Levin tarafından *Lucent Technologies*'de sürdürülmektedir.

SDL Türkiye'de hemen hemen tüm telekom yazılımı ile ilişkili olan kuruluşlarca (ASELSAN, BAŞARI,ALCATEL, NETAŞ vb.) kullanılmaktadır. Bu nedenle özellikle SDT ve benzeri SDL ürünleri ile yakın ilişki içinde olan NOKIA, ERICCSON,NORTEL gibi dev telekom şirketlerinin desteğinde yürüyen araştırmaları ve özellikle çeşitli Avrupa ülkelerinde yürütülen standardizasyon ve uygulama etkinliklerini izleyip Türkiye'deki şirketlere bilgi ve know-how aktaracak araştırmalara şiddetle ihtiyaç duyulacaktır. Bu tür araştırmalar bu projenin doğal devamı olarak görülebilir.

6 – EKLERİN LİSTESİ

Ek 1 COST 247 Preliminary Workshop

Ek 2 COST 247 Proposal

Ek 3 COST 247 Final Report

Ek 4 Levent Santralı TELETAŞ Projesi (EEEAG Yazılım 2 Projesi)

Ek 5 V. Levin, A. Sulimov, H. Yenigün and K. İnan, "Preliminary Concepts for a C++ Animator Design for SDL 92", published in Russian Academy of Sciences, Novosibirsk Informatics Institute Journal (EEEAG Yazılım 2 Projesi)

Ek 6 V. Levin, H. Yenigün and K. İnan, "An Implementation of SDL 92 Communication Semantics", Proceedings ISCIS X, Oct.30 - Nov. 1, 1995, Kuşadası, Turkey, pp. 457-464.

Ek 7 O. Başbuğoğlu, K. İnan, "Compiling SDL into the Finite State Specification Language Cospan", Proceedings ISCIS X, Oct.30 - Nov. 1, 1995, Kuşadası, Turkey, pp. 643-650

Ek 8 V. Levin, E. Bounimova and K. İnan, "An Interface Implementation Using a Compiler Specification Method", Proceedings ISCIS XI, Nov.6-8, 1996, Antalya, Turkey, pp. 385-394

Ek 9 V. Levin, E. Bounimova, O. Başbuğoğlu and K. İnan, "A Verifiable Software/Hardware Co-design Using SDL and COSPAN", Proceedings of the First COST 247 International Workshop on Applied Formal Methods in System Design, June 17-19,1996 Maribor, Slovenia, pp. 6-16

Ek 10 H. Yenigün, K. İnan, "A Verifiable Software/Hardware Co-design Using SDL and COSPAN", Proceedings of the Second COST 247 International Workshop on Applied Formal Methods in System Design, June 18-19,1996 Zagrep, Croatia, pp. 11-16.

Ek 11 K. İnan, H. Ural, "Efficient Checking Sequences for Testing Finite State Machines", Journal of Information and Software Technology, vol. 41 1999, pp. 799-812.

Ek 12 Tez Özetleri

EK 1

PROCEEDINGS OF THE PRELIMINARY WORKSHOP ON **COST 247** PROJECT ON VERIFICATION/VALIDATION AND TESTING OF SOFTWARE USING FORMAL SPECIFICATION TECHNIQUES

2 - 4 December, 1992, Ankara-Turkey

This proceedings consists of the revised contributions of the participants of the COST 247 preliminary workshop held in Bilkent Hotel, Ankara, during 2-4 December, 1992. The proceedings also contains a revised version of the original proposal submitted to and accepted by the TCT of COST. The revised proposal has taken into account the views, experiences and research interest areas of the participants of the workshop.

The purpose of the workshop was to establish a preliminary focus on the main areas of interest of the potential participants of COST 247 before the formal commencement of the project. Consequently the contents of this proceedings does not necessarily reflect a technically coherent set of original articles or presentations of a specific area of research, rather it consists of documents reflecting the areas of interest and experience and organizational structure of various institutions ranging from research departments of coorporations to public or private research institutions to universities within 11 different European countries including Turkey. Whereas some of the presentations emphasize details of technical results, most are structured in terms of institutional identifications in terms of past and current research experience and position statements vis-a-vis COST 247.

At the concluding session of the workshop the participants were explicitly asked what their research areas of interest were in the context of COST 247 . Below we itemize a resume of these areas based on the responses of the participating representatives of the institutions.

Preliminary Areas of Interest for COST 247

(1) Conformance Testing (special emphasis on real world examples and exploitation of systematic divide-and-conquer and use of black-to-gray box information contents)
(2) SDL '92 methodology (tool building, extensions, simulation and verification for object oriented specs., intermediate petri-net modeling etc.)
(3) Simulation (both logical and performance simulations using FDTs)
(4) Real-time specification (semantics and verification issues)
(5) Case studies to analyze and evaluate tools
(6) Complexity relief techniques for finite state verification
(7) Investigating relations between specification styles vs. verification and implementations, between verification vs. conformance testing.
(8) Specification assessment by reviews
(9) Hardware (VLSI) verification (symbolic or finite state)

The revised version of the COST 247 proposal reflects the areas of interest given above, but is formulated in a somewhat more general language to allow a space for other potential COST 247 participants interested in formal verification/validation techniques.

There has been a consensus within participants on the view that practical applications and reviews of techniques should reflect the problems of real life and not simplistic pet examples of restricted academic interest. This, of course, does not preclude in the research agenda of COST 247 untackled theoretical issues that may bear relevance and promise for future applications.

In terms of the use of existing specification languages interest was mostly on SDL - as the most commonly used tool in industry with a new version SDL '92 on the way - and LOTOS, although there was one representative that expressed interest on the use of ESTELLE as the formal language tool to develop performance simulation models from formal specifications. Again this does not preclude the use of other specification formalisms - such as special real time formalisms - in the research agenda.

The preliminary workshop on COST 247 is believed to have served its purpose. Namely, representatives from a wide ranging set of institutions and nations have converged on areas of research relevant to the application of formal techniques to verification/validation and related problems of software engineering. Thus the management committee, in its first meeting for COST 247, shall be in possession of a preliminary document and the anticipation of the members based on this document shall be of considerable assistance to set the detailed research agenda for COST 247.

February 4, 1993

Prof. Dr. Kemal İnan
Coordinator of the workshop and COST 247
Electrical & Electronics Engineering Dept.
Middle East Technical University
Ankara , Turkey

K2

GENERAL DESCRIPTION OF THE PROJECT
(revised version, February 4, 1993)

1 - Purpose

The purpose of the project is the coordination of national efforts to analyze ,classify and come up with new and efficient techniques and tools for concurrent software verification/validation , mostly in the context of existing standardized formal software specification languages ; apply and test the results on selected realistic problems in contemporary distributed communication architectures. The emphasis of the research shall be on the applicability of techniques/tools to realistic hence complex examples.

2 - Objective

Presently three formal specification languages, SDL, ESTELLE and LOTOS have become a part of international standards. Among these, SDL is the most commonly used one in communication industries and present research efforts are along the direction of industrializing the other two, especially LOTOS. In addition to these languages there is a considerable amount of specification languages and techniques for concurrent systems suggested by academic or other research bodies. An interesting example is the COSPAN specification language developed in AT&T for which a timing verification tool is recently implemented and is reported to be operational.

A crucial aspect of a formal description method is the technique/tool used to verify/validate a given formal specification against another given formal representation. On the other hand realistic and complex problems do not easily allow for exhaustive verification or symbolic proof systems used for verification and the usual industrial practice is to be content with logical simulation using reasonable simulation scenarios to verify a specification in some vague sense. Similarly generating conformance test sequences for complex examples is again based on suggestive scenarios of functional divide-and-conquer types. A principal objective of this research project is to investigate the possibility of tackling complex problems by studying the problem-specific, language-specific and style-specific aspects of existing simulation, verification and testing methods and suggest innovative and relatively more systematic techniques/tools - especially exploiting the decomposability properties - of validation for complex and preferably future-oriented architectures such as mobile and high performance networks. The results of the project will further the efficiency of software production cycle for complex distributed systems.

Most automated verification techniques used for complex formal languages are based on mapping the original specification into some intermediate model such as a finite state machine or a petri-net (or some finite state ω-automaton when liveness problems are of concern) , and apply the verification tool to the simplified model. Both the mapping involved and the complexity reduction techniques used for the simplified model are problem,language and specification-style dependent and require careful classification efforts to match the validation technique to the underlying problem. By focusing the research efforts to particular but representative selected examples, it may be possible to evaluate the efficiency of validation for the existing techniques, exhibit their limitations and suggest new and more effective approaches for the specific representative cases.

In addition to the review of validation and testing methodology for large realistic problems the research agenda shall also allow for the study of selected current theoretical and practical problems of interest. One such current research area is the area of real time semantics and verification. Concrete examples of industrial real time specification problems shall be of considerable value in order to differentiate between and evaluate numerous real time semantics and verification techniques that have appeared in the literature. Some of the recent research results (see remarks on COSPAN above) on real time, finite state verification using versions of temporal logic are already being applied to realistic examples and worth further investigation.

3 - Study Method

The study shall involve different specification and verification methods applied to representative examples involving specification refinement steps, timed specification (software or hardware), dynamical architectures such as in mobile networks, performance simulation from formal specifications etc. . The three year study shall consist of the following steps (diversity of topics may lead to seperate working groups) :
Phase 1 (year 1)
(a) Complex systems

- characterization of source material relevant to realistic verification and validation problems
- characterization of source material on conformance test generation with special emphasis on gray box assumptions
- selection of specification examples and, if possible, exchange of specifications written in a particular language (SDL, LOTOS) for the purposes

of joint review of styles, tools (e.g. new styles and/or tools for SDL '92) , verifiability and testability

(b) Real time systems

- characterization of source material on real time semantics and validation with an emphasis on practical verifiability
- choosing exemplary real time specification and verification problems (could be software or hardware).
- review of tools for real time verification

(c) Performance Simulation using FDTs

- Suggestions for a framework for using FDTs for stochastic timed (performance) simulation in the context of a specific language
- Design and implementation of a performance simulator based on the given FDT

(d) Other research problems in verification (hardware verification, use of theorem provers etc.)

Phase 2 (year 2)

(a) Complex systems

- Review and classification of specification style, verification, implementation, testing and other (possibly language or problem specific) kinds of problems that arise in complex examples and suggestions for improved methodology , techniques and tools
- Preparing the framework for applying improved methodologies or techniques or preliminary design of an improved tool for the kind of complex examples in question . This may be in the area of disciplined specification, verification, implementation (code generation) or conformance test generation

(b) Real time systems

- Experimentation of real time specification and verification using a selected language/tool on the problems selected
- Review of the theoretical and practical aspects based on the results of experimentation and suggestions for more efficient techniques/tools

(c) Performance Simulation using FDTs

- Design and implementation of a performance simulator based on the given FDT (Cont')
- Experimentation on the developed performance simulator on a given example

(d) Other research problems in verification

Phase 3 (year 3)

(a) Complex systems

- Choosing and experimenting on examples to apply the improved methodology and techniques and/or implementation of new tools for such purposes
- Review and exchange of the results obtained on complex systems

(b) Real time systems

- Design and partial implementation of verification tool(s) for general or specific kinds of real time specification problems

(c) Performance Simulation using FDTs

- Review and evaluation of the performance simulator and developing a unified theoretical framework for performance specification and verification

(d) Other research problems in verification

4 - Forms of Cooperation

The suggested cooperation within the project shall require the national delegates to :

- attend the meetings of the Management Committee : two to four meetings annually
- participate in setting up an active programme fitting with the objectives and the timing of the project
- coordinate the software requirements, tool exchange and experimentation examples for the national working groups.
- exchange the theoretical or empirical results of the national working group studies
- try to inform the national working groups on the results of the ongoing EC projects on formal methods

K 3

COST 247 Final Report

COST 247 was originally conceived of in 1992 and a preliminary workshop was held in Ankara, Turkey in December 1992. In this workshop there were representatives from industry, research organizations and universities from 10 different member countries of COST and the proceedings of this workshop contained position statements, including institutional settings and research interests of potential participants in the project. Broadly based on the outcome of this workshop the actual proposal was presented to COST during 1993 and was open to the signatures of member country representatives during 1993 after it was officially approved by COST. The inaugural meeting for the action was held in Brussels during December 1993 which has set the official starting date for the action.

The administrative body of the action, namely the management committee (MC), consists of the action chairman (K. İnan, Turkey), action vice-chairman (S. Budkowski, France) ,chair of working group 1 (WG1) on E-LOTOS (enhanced LOTOS) standardization activities (H. Garavel, France) , chair of WG2 on verification and analysis of formal specifications (J. F. Groote, Netherlands), chair of WG3 on conformance testing of communication software based on formal methods (K. Tarnay, Hungary), chair of WG4 on simulation and performance models using formal description techniques (P. Dembinski, Poland), and 2 delegates from each member country (the chair persons above are part of the 2 delegate teams from each country).

During the total duration of the action - four years after a one year extension request that was accepted in 1996 - the following activities took place :

(I) <u>MC meetings and regular technical presentations</u>

1st (inaugural) MC meeting : December 9,1993, Brussels, Belgium,
2nd MC meeting : 14,15 March, 1994, Utrecht, Netherlands,
3rd MC meeting : 19,20 September, 1994, Evry, France,
4th MC meeting : 9,10 February, 1995, Berlin, Germany,
5th MC meeting : 12,13 June, 1995, Warsaw, Poland,
6th MC meeting : 26,27 October, 1995, Budapest, Hungary,
7th MC meeting : 12,13 February, 1996, Madrid, Spain,
8th MC meeting : 17,18 June, 1996, Maribor Slovenia,
9th MC meeting : 4,5 November, 1996, Antalya, Turkey,
10th MC meeting : 18,19 June, 1997, Zagrep, Croatia,
11th MC meeting : 15,16 October, 1997, Stirling, UK,

(II) <u>Workshops</u>

(i) Preliminary workshop on COST 247, 10-13 December,1992, Ankara, Turkey
(ii) WG1 workshop on E-LOTOS , 19,20 July, 1994, Brighton, UK,
(iii) WG3 workshop on conformance testing, 7,8 September, 1995, Evry, France,
(iv) COST 247 *1st International Workshop on Applied Formal Methods In System Design* , 17-19 June, 1996, Maribor, Slovenia,
(v) WG3 workshop on conformance testing, 9-11 September,1996, Darmstadt, Germany
(v) COST 247 *2nd International Workshop on Applied Formal Methods in System Design* , 18 - 19 June 1997, Zagreb, Croatia.
(vi) WG3 workshop on conformance testing, 8-10 September, 1997, Cheju, Korea

(III) <u>Publications</u>

(i) Proceedings of *1st International Workshop on Applied Formal Methods In System Design* , June 1996, Maribor, Slovenia,

(ii) Proceedings of *2nd International Workshop on Applied Formal Methods In System Design* , June 1997, Zagrep, Croatia,
(iii) Special COST 247 issue of the journal *Science of Computer Programming, vol. 29, No. 1 & 2 , July 1997,* under the theme "Methods of Software Design: Techniques and Applications" edited by J. F. Groote (WG2 chairman) and M. Rem.

(IV) <u>Short Term Missions</u>

(i)
Host : Prof. Ken Turner, Stirling University, UK.
Visitor : Prof. G. Csopaki, Budapest University, Hungary
Date of visit : 3-10 August, 1996
Nature of the work : Modeling digital logic in SDL and HW/SW codesign
(ii)
Host : Steve Schneider , Royal Holloway University , London, UK
Visitors (COST 247): Guy Leduc (Liege University), Juan Quemada (Technical University of Madrid), Luis Llana (University of Complutens, Madrid),
Other participants : Lac Leonard (University of Liege), Gualberto Garbay (Technical University of Madrid), Alan Jeffrey (University of Sussex), Jim Davies (University of Reading), Jeremy Brians (University of Reading)
Date of visit : November 13-18, 1994
Nature of work : Converging towards a real-time extension of LOTOS
(iii) Host : G. Leduc
Visitors (COST 247) : Hubert Garavel (INRIA, Rhones-Alpes), Alan Jeffrey (University of Sussex)
Other participants : Mihaela Sighireanu (INRIA Rhones-Alpes), Ricardo Pena (University of Complutens, Madrid) , Charles Pecheur (University of Liege)
Date of visit : April 22-25,1996
Nature of work : Setting down the basis for the new data types in E-LOTOS

(V) International Conferences with joint COST 247 participation

IFIP PSTV 13-15 June, 1995, Warsaw, Poland

IFIP IWTCS ,7-10 September, 1995, Evry, France.

Eleventh International Symposium on Computer an Information Sciences(ISCIS XI) 4-7 November, 1996 , Antalya, Turkey.

IFIP IWTCS ,8-10 September, 1997, Cheju, Korea.

(VI) Standardization Activities

E-LOTOS standardization activities in former CCITT and present ITU

The rest of the final report are the abstracts of all the papers presented at the technical sessions and/or workhops of COST 247.

EK 4

Description

of the joint project of TÜBİTAK SR&DC with ALCATEL-TELETAŞ

Review of a Design for Centrex - Levent System Using SDL

The project involves the detailed design of the "Centrex" facilities to be integrated into the existing rural switch Levent designed and produced by ALCATEL-TELETAŞ. The project objectives are stated below.

(1) The SDL specification of the Centrex features integrated into the existing Levent switch shall be generated. The SDL specification of the features that are relevant to Centrex and that already exist in Levent shall be covered in the final integrated SDL specifications. In case the complete detailed design SDL specifications of the Centrex functions require beyond a 1 man-year effort either a relevant subset shall be covered or the necessary additional human resource shall be recruited subject to the availability of such human resource and the mutual agreement between TELETAŞ and TUBITAK.

(2) The entire detailed design shall be analyzed (simulation, verification) making use of the tool SDT available in TUBITAK Software Research and Development Center. The outcome of this study shall be reported to TELETAŞ.

(3) A separate part of the final report shall consist of methodological considerations. In particular throughout the process of writing and analyzing SDL specifications all the bottlenecks and/or suitability of different specification styles or approaches shall be taken into consideration in order to come up with concrete suggestions related to general software methodology that yields maximum efficiency in speed, quality and flexibility (modularity etc.) of the final software product.

(4) The long term mission of the project is methodological. In particular this project is viewed to be an initial industrial exercise to assess the feasability of a new methodology for software production based on plug-in modularity as in hardware design. Current research efforts in industrial communication software production have focused on the use of object oriented techniques in order to achieve modularity, code re-usability and prototyping. The new standard for SDL, namely SDL '92, is endowed with object oriented facilities and testing the use of these facilities in the context of concrete industrial design examples is of substantial interest to communication and software companies as far as speed and efficiency of software production is involved. In TUBITAK a seperate research team is currently implementing an SDL'92 simulator and is familiar in depth with the associated features of SDL'92. Therefore although the specifications and analysis for Centrex shall be implemented making use of the existing SDL'88 language, effort shall be spend on the design of the object structure (i.e. class hierarchy and object library) for the Centrex example in terms of the concepts of SDL'92. A current European research project that parallels the vision above - albeit a closed one - is an EUREKA project that focuses on

generating the necessary libraries and the additional process objects in the context of the language C++. The only partners in this industrial project are Bristol HP and Grenoble HP.

In order to meet its objectives the following procedure shall be adopted throughout the project :

(1) There shall be at least 2 points of synchronization between TUBITAK and TELETAŞ in the project:

(i) After the completion of the top level architecture design by TELETAŞ the outcome shall be disseminated to the TUBITAK research group and from there on both parties shall pursue the detailed design independently. The estimated time for this point of synchronization is the end of June 1994.

(ii) After the completion of the detailed SDL design specifications the parties shall exchange their versions and a common SDL specification shall be adopted as a result of a mutual consent based on their experiences and design considerations. From there on there shall only be a single SDL specification for further analysis by TUBITAK and for implementation by TELETAŞ. The estimated time for this phase of synchronization is November 1994.

(2) In addition to these two points of synchronization TUBITAK shall produce 3-monthly reports on the progress of the work carried out and a final conclusive report that contains all the relevant findings and results related to the project summarized above.

(3) Throughout the project communication of the parties shall be established through an operational communication link (e-mail+ftp, etc.). All the technical discussions as well as software and/or document transforms shall preferably make use of this link.

(4) TELETAŞ shall supply TUBITAK with the technical details of the Levent exchange and other relevant information whenever necessary.

(5) Since the project shall be carried out in the context of the international COST 247 project TUBITAK shall make best use of exploiting the international contacts (workshops, short term missions, etc.) according to the possible demands of TELETAŞ.

The project shall be a 1 man-year project starting as of June 1, 1994. Any decision to extend the human resource beyond the 1 man-year limit is subject to the mutual agreement between TELETAŞ and TUBITAK.

EK 5

PRELIMINARY CONCEPTS FOR A C++ ANIMATOR DESIGN FOR SDL 92 [1]

V. Levin, [2] A. Sulimov [3], H. Yenigün and K. İnan [4]

Abstract

Preliminary considerations for a C++ animator design for SDL 92 are presented as the starting point for the corresponding project. The design involves compiling SDL 92 into C++ code. C++ has been chosen as the target code in a hope to capture the object oriented features of SDL 92 in a straightforward manner. The paper emphasises this point just after the top level considerations.

Keywords : Communication Software , Specification Languages, Compilers, Object Orientation

1 Introduction

SDL (Specification and Description Language) is a formal specification language which is a joint CCITT-ISO standard [1, 2]. Originally conceived for telephony network specification in early 1970ies it was later extended to incorporate data communication systems. Recently the SDL 88 version was updated and a new standard called SDL 92 was issued. SDL 92 incorporates new features that gives it a strong object orientation flavor and for that reason hopes have been invested in it as a productive CASE tool to be used in communication software methodologies .

Possibly owing to its simplicity as practiced by the engineers-in-the-field, SDL is a widely used formal specification language in large communication industries, especially compared to its other two rival formal specification languages LOTOS [3, 4] and ESTELLE [5, 6] which are also CCITT-ISO standards. This is one pragmatic reason for focusing on a software platform to be designed around SDL. Another reason lies in the nature of the complexity of modern communication systems. Whereas rapid developments in electronic hardware and communication media technologies such as fiber optic reduced the costs of mass standardized produced components. The same cannot be stated for the residing software in these components. The complexity of the software for such highly distributed and heteregenous systems are mostly due to distributed communication architectures, heteregenuity of interfaces, existence of multivendor hardware and software environments etc.

[1]Research supported by the Scientific and Technical Research Council of Turkey (TÜBİTAK) Projects EEEAG Yazılım 2 and COST 247

[2]presently with the Software Research and Development Center (SRDC) of TÜBİTAK, Ankara, Turkey, on leave from the Keldysh Institute, Moscow, Russia.

[3]During March - August 1994 he was with SRDC, TÜBİTAK , presently he is at the Institute for Informatics, Russian Academy of Sciences, Novosibirsk, Russia.

[4]both are with the SRDC,TÜBİTAK, Ankara, Turkey.

Producing and maintaing software for such systems can be both time consuming and subject to unexpected errors. A formal tool like SDL 92 possessing object oriented features such as data encapsulation, code re-usability etc. is hoped to answer some of the key needs of current software technologies.

The present paper describes an initial effort along the direction explained in the paragraph above. Exploiting the similarity in the object oriented features of SDL 92 and the language C++, results of a preliminary compiler design are reported. As a first step in establishing a software architecture around SDL 92 it was decided to implement the *animator* together with the SDL graphical editor, GR/PR and PR/GR converters and other graphical or similar tools required for practical operationality. Future steps shall incorporate building tools such as verifier/validator, code generator for specified lower level platforms, test sequence generator, TTCN [7] editor etc.

An animator of a distributed system is a software that has the capability of executing possible sequences of execution of a communicating system. In particular an animator based on SDL generates any feasible sequence of executions of a distributed system with concurrently operating and communicating components formally specified in SDL. Unlike sequential software, errors in concurrent software usually stem from unimagined but possible execution orders of its concurrently operating components. An animator is the simplest validation tool that supports an operational confidence on the specified software by executing typical message exchange scenarios on the specified system. A thorough verification can be done either by a computer aided symbolic proof system or via an exhaustive finite state verifier provided that the computational resources can cope with the system complexity. In terms of the logical priority of the software platform mentioned above, the animator is higher up on the hierarchy since tools like verifier, test sequence validator or code generator shall all make use of the animator in order to generate all or desired execution sequences. Structurally speaking these advanced tools have to make use of and be based on an intermediate (common) core data (process) representation which also forms the backbone of the animator [8].

In the next section we briefly describe the characteristic features of the language SDL 92. In section 3 we present an outline of the SDL-isomorphic C++ structures that capture various aspects of SDL 92 constructs and additional C++ constructs that are used for the purpose of animation and maintaining the synchronization semantics of SDL SDTdefinitions.

2 An Outline of SDL 92

In this section we present a brief guided tour of SDL 92. The level of details is no more than to give the reader a flavor of the language and the reader is referred to [9] as a detailed and well-guided source and [2] as the formal standard definition of SDL 92.

SDL has two isomorphic syntactical definitions : graphical and textual. The graphical syntax SDL/GR is suggestive of the architecture of the communication system specified

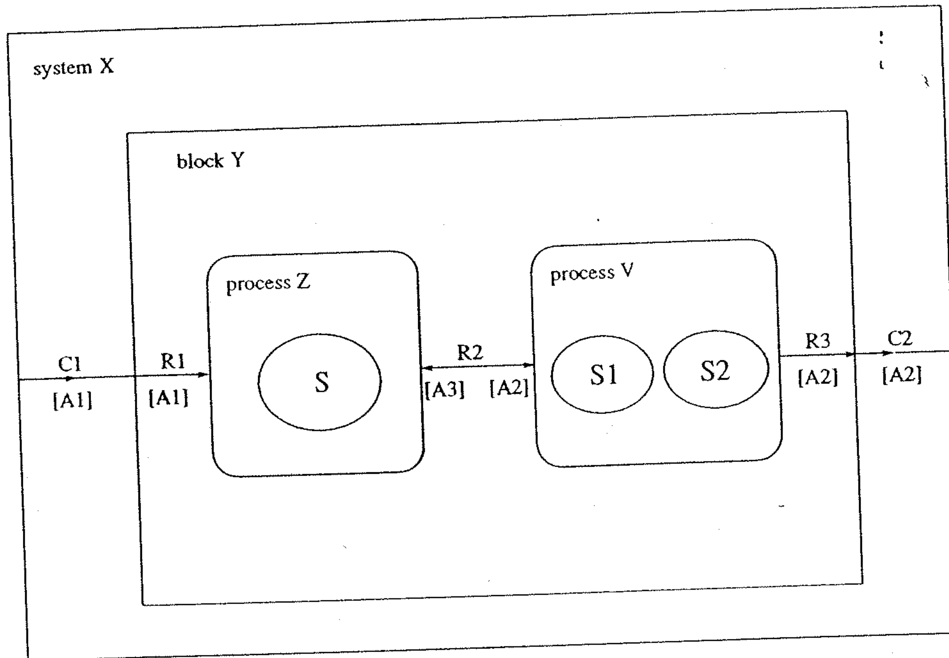


Figure 1: The SDL specification X

and hence much more preferred by the engineers-in-the-field as a tool for specification. An SDL system is composed of SDL *blocks* as seen in Figure 1 . Blocks can be nested and they usually model geographically coherent functional units of a system. Blocks are connected via directed communication *channels* which model communication links connecting geographically seperated communication units. In SDL a communication channel is defined as a FIFO queue. Unless the channel is specified to be delay-free, the underlying FIFO queue is assumed to be infinite whereas a delay-free channel is a zero length queue forcing spontaneous synchronization on its two end points. Each channel is allowed to carry a set of *signals*. A signal name signifies a communication message of a specific type with possible parameters of a desired data structure. Time delays occuring in channels are arbitrary so that SDL semantics nondeterministically allow for all execution sequences corresponding to delays within channels that occur arbitrary in space (different channels) and time (different instances).

The innermost unit of a possibly nested hierarchical block structure is called an SDL *process* . Processes within a block are connected via *signal routes* which are delay-free channels. Every process in SDL has a *Pid* (process identification). Processes can dynamically be spawned by other processes. In the Figure 1 the block V is nested inside the system X. Channels from and to the *environment* are specified by $C1$ and $C2$ whereas signal routes $R1$,$R2$ and $R3$ connect processes Y and Z within the block V to each other and to the external channels. Signal name lists for each of the channels and the signal routes are specified, for example channel $C1$ carrying the signal $A1$ etc.

Notwithstanding some exceptions such as dynamic process creation, procedure calls etc., processes in SDL can be thought of as generalized finite state machines. Part of the global

state of an SDL process - see input queue below - consists of a location (generalized state) within a generalized finite state transition diagram together with the current value of its internal variables. Each process has a distinct infinite input signal queue that receives signals from other processes via signal routes or channels. Thus the current contents of the input queue constitutes the remaining part of the global state of the process.

The graphical syntax that defines an SDL process is a program flow diagram of communication (read or write) or internal execution sequences. A communication execution is either reading a signal from the input queue of the process or sending a signal to a signal route or a channel that is eventually destined to another process within the system. A simple internal execution is typically an (task) assignment statement or a logical branching. More complex executions involve spawning a child process or calling a procedure. An SDL procedure defined within an SDL process is like a separate SDL process that shares the input queue of the process and may share the variables defined within the calling process. The calling process is frozen until the procedure process terminates and returns to the calling instance of the original calling process.

SDL 92 differs from its previous version SDL 88 in a number of respects. First, *instances* of SDL *entities* (e.g. blocks and processes) are differentiated from *types* of entities. The graphical symbols used for instances and types of entities also differ. In the terminology of object orientation types correspond to classes whereas instances correspond to objects. By defining only the additional transitions for a given process type, new subtype processes can be defined using the inheritance mechanism. Overriding of methods (transitions) within the subtypes is also allowed by using *virtual* markings for the transitions of the parent process. All this is hoped to make future life easy for a communication software department by reducing the software specification and design methodology to a plug-in type of hardware methodology by an appropriate use of a communication oriented class hierarchy and object library.

A useful SDL construct is a service block defined within a process. A process can be conceived to consist of one or more service blocks. A service block within a process is in fact like an independent process except that it shares the common input queue and some of the variables of the process. The signals consumed (read) by different service blocks within a process are disjoint and therefore no contention for input can take place. Service blocks operate in a disciplined interleaving order whereby every service block moves from one SDL state (a name given to the states where signals from the input queue are read and consumed) into another while all the remaining service blocks are kept frozen at their respective SDL states. The moving service block is uniquely determined by the contents of the current and common input queue. Service partitioning is useful in designing processes that actually perform concurrent tasks using disjoint sets of messages and sharing the communication buffering resources (input queues).

Finally it is worth noting that SDL has a complicated semantics principally because of its complex and difficult-to-implement synchronization mechanisms. To take an extreme example, if the type of a destination process and a set of channels and/or signal routes are

given then SDL semantics demand that this signal reaches to one such process instance of the type specified nondeterministically provided that currently such an instance exists and a link consisting of channels and signal routes from the outputting source to the instance in question exists that make use of at least one element of the given channel and signal route set [5] .

As to be witnessed in the next section the SDL 92 to C++ translation is not a simple conversion of class and object structures of SDL 92 into the corresponding object oriented structures of C++ and requires intermediate control process structures to meet the animation requirements and to maintain conformance to the SDL semantics.

3 C++ Structure of the SDL 92 Animator

Underlying the conceptual intermediate data structure that in turn is translated into C++ structures is a CSP-like process algebra [6] called ERPAL [10, 11]. The abstraction involved in converting SDL programs into C++ structures therefore consists of two steps : the first step is converting the entire operationality of the SDL animator into a new process expression in ERPAL that incorporates auxiliary control processes as well as translations of the original SDL processes and the second step involves converting the ERPAL processes into C++ structures. However because ERPAL has no object orientation features and the intermediate ERPAL steps above can be dissolved within the final transformation into C++ it is preferable to explain the direct translation primitives into C++ and refer to the intermediate process abstractions whenever relevant.

The C++ animator for SDL 92 consists of two parts. The first part consists of general control processes and structures that are common to all SDL 92 programs. In particular there is the process called ControlManager that manages the entire animation steps, another process ShowManager controls the visualization of the animation. Other auxiliary control structures are included in a support library of C++ classes.

The second part of the animator consists of the classes that correspond to the actual block and process types, channels, signal routes etc. of the actual SDL 92 program. The class that corresponds to a process type in C++ - which in fact is an ERPAL process - is more general than and somewhat different from an SDL process.

More specifically for every process type defined in SDL 92 (there may be no current instance of it !) there exists an ever-present ERPAL process represented by a C++ class called the *PiSet* manager or simply PiSet for that type. This process is responsible for creating, deleting and in general book-keeping (such as keeping a list of the Pid's of all the

[5] In the opinion of the authors of this article this is a very peculiar and a global routing constraint which should fall within the responsibility of the design specifications rather than the synchronization semantics of a language.

[6] An important point to note here is the kind of synchronization semantics used by ERPAL which is *rendezvous* type that can model the buffered queue synchronization of SDL.

live instances) all the instances of the processes of that type. In particular all messages terminating in and emanating from instances of a given process type go through the PiSet process of the given type. The principal use of this ERPAL control construct is to satisfy the synchronization semantics of SDL.

A plain SDL process type consisting of a single service in SDL 92 transforms into two ERPAL processes operating in parallel corresponding to two C++ classes. One of these is the actual body of the SDL process and the second one is a process called an *input port* process that manages the input queue of the process. If the process consists of several service blocks than each service block is modeled by a distinct ERPAL process communicating (synchronizing) with the common input queue manager process.

Every SDL channel is also modeled as an ERPAL process with a single variable of generic queue type. All the processes above are related to each other via repeated use of a universal parallel synchronization operator of ERPAL. Synchronization in the SDL context has a simplifying aspect : no process can be blocked on an outputting operation and for every act of outputting there corresponds a unique input operation of filling a channel slot or an input queue slot of a given process - or, of course, losing of a message occurs if no process can synchronize with the output operation. In other words mutual consent for synchronization is not required as in a general rendezvous type of synchronization and every output transition uniquely determines the outcome of the corresponding synchronization move.

Similarly each SDL procedure defined within a process is modeled by a separate ERPAL process that interacts with the calling process via a recursive use of ERPAL sequential (or stack) operator. This operator is similar to Hoare's sequential operator ";" or the sequential operator ">>" of LOTOS [4].

This completes the outline of the process based description of the SDL 92 animator. Before we present the C++ structures for the constructs above we briefly mention two different, yet important, hierarchical structures in SDL 92 to C++ conversion. The first hierarchy is the architectural hierarchy of the communication system specified. At the top of this hierarchy is the entire system represented by the SDL system block as in Figure 1. Nested blocks and channels and signal routes within blocks represent the functional hierarchy of the system specified. The lowest atomic elements of this hierarchy are the SDL processes or more precisely service blocks or active procedures incorporated in the processes. The manner in which this hierarchy is captured in the C++ image consists of nesting of classes within classes. Referring to the example given in Figure 1 the C++ classes corresponding to the process type Y and signal routes $R1$ to $R3$ are defined within the C++ class corresponding to the block type V. The explanation for the second hierarchy , namely that of the C++ classes is postponed to the next section.

4 Representation of SDL 92 constructions in C++

4.1 Functional and Class Hierarchies of the C++ Code

The C++ structure corresponding to an SDL 92 program consists of two parts :

1. an input–dependent part that constitutes the C++ images of SDL constructions,
2. an input–independent part that is necessary for the purposes of animation.

The input–dependent part produced by the compiler preserves the first kind of systemic hierarchy explained in the previous section. It does so by mapping the nested hierarchical SDL entities [7] into a similarly nested set of C++ classes. Observe that the SDL hierarchy explained in the previous section can be visualized as a tree with nodes as SDL entities and branches as the nesting relations. The compiler can then be viewed as a transformer of this tree into its isomorphic image tree where the transformed nodes are the corresponding C++ classes. There are some exceptions to this rule. For example variables, which are entities that are nested inside a process which is another entity, are not mapped into C++ classes. Rather they are defined as variables within the nesting C++ class.

We shall call SDL entities that are mapped into C++ classes SDL *units*. Among these SDL units we further make *active* and *passive* unit distinction. An active SDL unit is one that correspond to an ERPAL process. Namely, an entity that has a property of evolving in time (i.e. it has a state). System, block, channel, signal route, process, service, procedure and timer constitute active SDL units. Passive SDL units, on the other hand, are signals and user defined data types.

Let an SDL unit be referred to as a self–instantiated unit if the instance of its creation always coincides with that of the unit that embraces it. In this sense system, block, channel, signal route and service are self–instantiated SDL units whereas, in general, process, procedure, timer and signal are not self–instantiated units. A self–instantiated SDL unit is statically instantiated if it is created in the beginning of the animation process. In this sense with the exception of service, all self–instantiated units are statically instantiated. In case of a self-instantiated unit Y, the C++ object representing its instance is created and bound to the corresponding pointer PY within the constructor of the embracing C++ class CX.

Figure 2 below explains the next hierarchical structure, namely the class (inheritance) hierarchy mentioned in the previous section. It depicts the class hierarchy structure of the C++ image of SDL programs. Here the classes named *base* classes correspond to abstractions of concrete C++ classes corresponding to concrete SDL units.

Figure 2 contains only the image classes of the SDL units. The set of all C++ classes generated by the compiler are greater than this image set. In particular there are two

[7] In SDL the term *entity* is used for SDL constructs such as system, block, process, signal, variable etc.

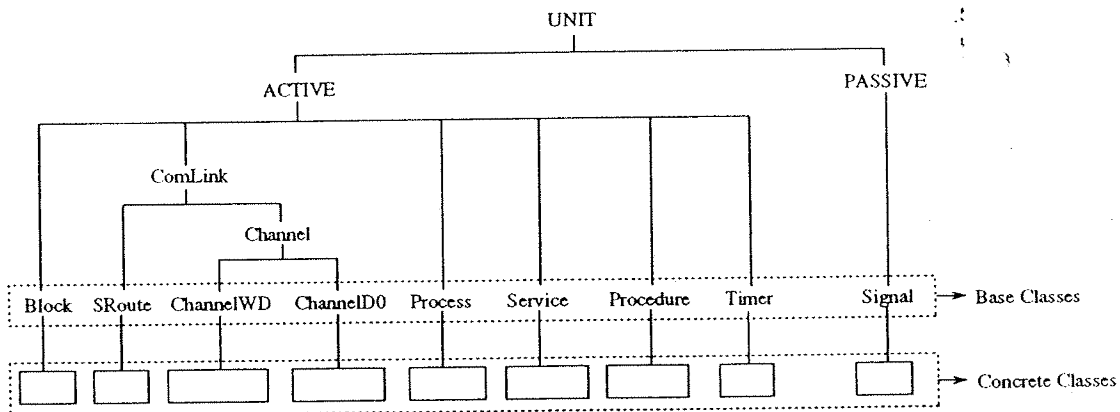


Figure 2: Class Hierarchy

important C++ classes on the level of the base classes that are generated for the purposes of animation dynamics. These correspond to the following process types mentioned in section 2 : input port process for managing the input queue of a process and PiSet process for managing the instances of an SDL process type.

The top level class in Figure 2 has two methods as defined by the C++ code below :

```
typedef unsigned int card;
typedef card Idcode;
class Unit {
 public:
  virtual Idcode   idcode () = 0;
  virtual char*    name () = 0;
}; // Unit
```

These methods represent identification and naming of units. The method name returns a string given in the SDL program as the name of a unit. The method idcode returns a cardinal number which uniquely identifies the unit considered. Both methods are concretized in the concrete classes derived from the base ones. The example below shows an outline of the C++ code corresponding to the SDL system given in Figure 1.

Example 1

```
class X : public Block {
 public:
  Idcode   idcode () { return 0; };
  char*    name ()   { return "X"; };
  class A1 : public Signal {
   public:
    Idcode   idcode () { return 1; };
```

```
    char*    name ()    { return "A1"; };
    ...
  }; // A1
  class A2 : public Signal {
   public:
    Idcode   idcode () { return 2; };
    char*    name ()    { return "A2"; };
    ...
  }; // A2
  class Y : public Block {
   public:
    Idcode   idcode () { return 3; };
    char*    name ()    { return "Y"; };
    class A3 : public Signal {
     public:
      Idcode   idcode () { return 4; };
      char*    name ()    { return "A3"; };
      ...
    }; // A3
    class Z : public Process {
     public:
      Idcode   idcode () { return 5; };
      char*    name ()    { return "Z"; };
      ...

    }; // Z
    ...
  }; // Y
  ...
}; // X
```

4.2 Animation methods

Animation of an SDL program makes use of two animation methods: ask and go. The ask method simply implements searching for all the next events that may be executed in an SDL entity whereas the go method actually implements the execution of the event given as the argument of go. C++ objects that make use of these methods are the animated objects which correspond to SDL active units and process instances sets. So, the methods ask and go are virtual within the base classes ActiveUnit and PISet. In the tree of animated objects the leaves may be services, channels or active procedures.

The ask and go methods propogate right through the tree of animated objects. More precisely in the case of the ask method applied to node of the tree the method is recursively passed down to the children until the leaves and the result is accumulatively passed on to the original node. Once the user selects an event out of the possible next events offered by

the ask method it first locates the object where this event lies and executes the method go at that object. Depending on the possible mode of use, these methods can be invoked by the user explicitly or can be invoked automatically and subordinate to some execution directive - such as execute all invisible events until the next visible (communication) event.

4.3 Implementing Visibility and Scoping

Implementing the visibility and scoping rules of SDL are based on the functional hierarchy of the nested C++ classes mentioned earlier. In SDL system, block, process, service and procedure are referred to as *scope units* and the name of an entity defined in a such a scope unit can be visible not only in this unit but also inside the nested scope units. As explained above, in our C++ representation, SDL units are represented by classes whereas other SDL entities (variables, synonyms) are represented by C++ variables. They are declared (as data members) inside those C++ classes which represent the SDL scope units defining the corresponding SDL entities (variables, synonyms). Unfortunately whereas the hierarchy of the nested C++ classes preserves and therefore automatically implements the visibility and scoping semantics corresponding to SDL units the same is not true for C++ variables corresponding to SDL entities given by variables and synonyms. Since the visibility and scoping semantics for the variables defined within C++ classes do not match the similar semantics for the SDL entities that they represent, additional measures are required for a correct implementation.

Referring to Figure 1 above and using the SDL rules the signal names A1 and A2 are visible on the top level of system X and inside its inner block Y as Y does not redefine these signals. On the other hand the signal name A3 is visible inside block Y and, in particular, inside process Z, but not visible outside block Y since the definition of this signal is encapsulated by block Y. Correspondingly referring to the code given in Example 1 above, the signal classes A1 and A2 are visible on the top level of system class X and within block class Y. Signal class A3 is visible inside block class Y and, in particular, inside process class Z, but not outside Y.

The remark above illustrates how the SDL scoping and visibility semantics follows by a straightforward C++ implementation for SDL units. On the other hand as expressed in the paragraph above this approach does not solve the problem for SDL variables and synonyms. The access to a variable declared in the embracing object and used in a nested object is provided inside this nested object by making use of a pointer *up* (in the C++ sense) pointing to the embracing object in question. Consider Example 2 below where a service S accesses a variable v declared in the embracing process Z. As seen in the example a pointer up defined in class S is initialized by address of an object of the embracing class Z. This initialization is done inside the constructor S : when invoked in constructor of Z it takes as its argument construction *this* (in C++ sense) which keeps address of the embracing object of class Z.

Example 2.

```
class Z : public Process {
 public:
  Idcode   idcode () { return 5; };
  char*    name ()   { return "Z"; };
  int v;
  ...
  class InputPortZ : public InputPort<2> {
   ...
  }; // InputPortZ
  class S : public Service {
   protected:
    Z * up;
   public:
    Idcode   idcode () { return 6; };
    char*    name ()   { return "S"; };
    S (Z * u) : up(u) { ... } ;
    ...
  }; // S
  S * s;
  InputPortZ * input_port;
  Z () { s = new S (this); input_port = new InputPortZ; };
  ...
}; ///////////////////////////////////////////////////////////////////// Z
```

4.4 A Process and a Service

In SDL, a process may be divided into services sharing its input port and some variables. Another variant is a process without services. In the first case, all the process states and transitions are written inside its services, in the second case – directly inside the process. In order to simplify C++ representation, we reduce a process without services to a process with one service.

Below is a short resume of the interaction between a process input port and the services of the process. In accordance with SDL , the services of a process partition the incoming signals into disjoint groups so that a signal put into the process input port can be consumed by exactly one of the services of the process. Only the STATE nodes of services are allowed to consume signals and atomicity of STATE node-to-STATE node transition is preserved by freezing all other services at their STATE node while the active service completes its transition to its next possible STATE node. Since the services of a process use disjoint sets of signals there is no ambiguity or contention as to which unique service deals with (consumes or saves) the signal at the head of the common input queue.

In SDL an INPUT branch may have a priority which means that the associated signal with this branch is consumed first, where all the signals without priority that lead this

signal in the queue are saved. This is equivalent to the SAVE primitive of SDL where the signals saved are those without priority in front of the priority signal.

As explained in the section above, an SDL process is an SDL unit and hence is represented by a C++ class. This class contains nested classes which represent the services, the input port of the process and timers, procedures, signals and sorts (if any), i.e. those SDL entities which are defined within the process. Example 2 above exhibits an outline code for the C++ class representing the process Z. A service is also an SDL unit and hence represented by a C++ class, which contains nested classes representing procedures and sorts, if any, defined within it.

The input port of a process Z is represented by class `InputPortZ` coupled with pointer `input_port` which provides access to an object of this class and is declared within the C++ class representing process Z. The class `InputPortZ` is specific for process Z, because it deals with the set of its valid input signals specific for the process. On the other hand, class `InputPortZ` is derived from the base class InputPort given below and inherits its methods.

```
template < card N >
class ProcessInput {
 protected
  list <Signal> queue;
  enum SignalStatus {ACCEPTABLE, WITH_PRIORITY, SAVABLE, REMOVABLE};
  SignalStatus signal_status [N];
 public:
  void input ( Signal * );
  Signal * consume ( );
  void acceptable ( card );
  void with_priority ( card );
  void savable ( card );
  void reset_signal_status ( );
  ...
}; // ProcessInput
```

A short description of the InputPort class methods follows. Method `input` is invoked with a pointer to a signal as the argument, and places the pointer into list `queue` which keeps a queue of incoming signals. In class InputPort there is also array `signal_status` whose elements are associated with the elements of the set of valid input signals. A value of each array element encodes the current status of the signal that the element associates with – a signal may have status `ACCEPTABLE`, `WITH_PRIORITY`, `SAVABLE` or `REMOVABLE`. Array `signal_status` is filled by the services, namely, when a service reaches a STATE node. The contents of list `queue` and array `signal_status` provide necessary information for method `ask` to detect whether the process instance envisaged can consume a signal from its input port. A service consumes a signal using the method `consume`.

```
class Service : public ActiveUnit {
```

```
public:
  ...
protected:
  EventTbl events;
  card     next_node;
  card     node_entry;
}; // Service
```

The nodes of SDL services are in fact the current states of the service objects. The methods ask and go applied to objects involves computation based on the current states of those objects. Whereas the method ask returns possible transitions from the state (node), the method go actually forces this transition to occur into the next state. Example 3 below exhibits an outline code for the C++ class representing the service S nested inside the process Z.

Example 3.

```
class S : public Service {
  public:
   Idcode idcode () { return 6; };
   char*    name () { return "S"; };
  protected:
    Z * up;
    virtual void go_cont1 () {};
    virtual void ask_cont1 () {};
    virtual void wait_go_cont1 ( Signal * ) {};
    int x_v;
    void   START_go    () { next_node = 1; };
    void   START_ask   () { events.init_1 ( idcode(), next_node, START_EV ); };
    void   ASSIGN_go   () { x_v = 0;  next_node = 2; };
    void   ASSIGN_ask  () { events.init_1 ( idcode(), next_node, INTERNAL_EV ); };
    void   wait_go     ();
    void   wait_ask    ();
    void   OUTPUT_go   ()
    void   OUTPUT_ask  () { events.init_1 ( idcode(), next_node, OUTPUT_EV ); };
    void   STOP_go     ()
    void   STOP_ask    () { events.init_1 ( idcode(), next_node, STOP_EV ); };
  public:
    S ( Z * u ) : up (u) {
      events.init_empty;
      next_node = 0;
      node_entry = 1;
      x_v = default_G.int_t;
    };
    void go ();
```

```
    void ask ();
  }; // S
```

In Example 3 variables events, next_node and node_entry belong to the base class Service. They represent respectively a set of events possible for this service, the number a node which will execute next, just as the service goes, and the number of entry into this node which is used for STATE nodes. Variable x_v represents local variable x of the service S. The four variables are initialized by constructor S. The structure object default_G contains default values for SDL data types; in particular the field int_t gives the default value of data type int.

The control flow graph of a service is implicit in the implementation of the go method updates the node counter next_node. The go and ask methods of the service make use of this node counter to choose the next node: see Example 4 below.

Example 4.

```
void Z::S:: go () {
  switch (next_node++) {
   case 0 : START_go();  break;
   case 1 : ASSIGN_go(); break;
   case 2 : wait_go();   break;
   case 3 : OUTPUT_go(); break;
   case 4 : STOP_go();   break;
   default: go_cont1();
  }
};

void Z::S:: ask () {
  switch (next_node) {
   case 0 : START_ask();  break;
   case 1 : ASSIGN_ask(); break;
   case 2 : wait_ask();   break;
   case 3 : OUTPUT_ask(); break;
   case 4 : STOP_ask();   break;
   default: ask_cont1();
  }
};
```

Example 3 demostrates that _ask methods fill the table called events with an element representing an event by a triple, namely service identifier code, the node counter and an event type. _go methods given in Example 3 are the simplest; yet a more complicated one is given in Example 5 below.

Example 5.

```
void void Z::S:: wait_go () {
  Signal * s;
  ...
  switch ( node_entry ) {
   case 1:
     node_entry = 2;
     up -> input_port -> reset_signal_status ();
     up -> input_port -> acceptable ( A1.idcode() );
     up -> input_port -> acceptable ( A3.idcode() );
     break;
   case 2:
     number_of_entry = 1;
     s = input_port -> consume ();
     if ( s -> idcode == A1.idcode() ) next_node = 3;
     if ( s -> idcode == A3.idcode() ) next_node = 4;
     wait_go_cont1 ( s );
   }
}; // wait_go
```

In Example 5, the node method wait_go corresponds to the STATE node wait given in Figure 3. The implementation consists of two parts into which the two entries are connected via a switch statement. The first part conveys to the process input port information about status of signals. The second part consumes a signal from the process input port by calling method consume of input_port and makes its respective transition.

The role of methods go_cont1, ask_cont1, and wait_go_cont1 are explained in the subsection below.

4.5 Representation of SDL Object-oriented Features

In SDL 92 there are three object-oriented features: generic types with formal context parameters, inheritance, and virtuals. Each of these are in a part captured by C++ object-oriented features either in a straightforward manner or by indirect simulation schemas.

In simple cases SDL generic types are directly captured by C++ generic classes with template parameters. Processes, procedures, signals, timers and sorts as formal context parameters are represented by template classes. We can thus manage not only independent formal context parameters but also a typical formal constraint concerning signals with data decribed in [9] as follows

process type pass_signals

```
    < signal a ( atype ); newtype atype >
```

This generic process type is mapped to the C++ class below.

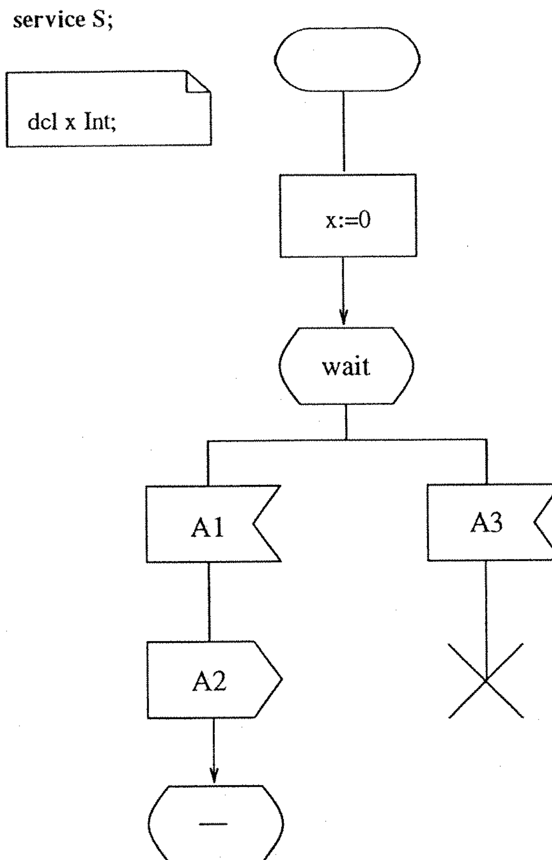


Figure 3: Service S

```
template < class a, class atype >
class pass_signal {
  ...
}; // pass_signal
```

Inside this class the operator `new a(d)` where `d` is a variable declared of the `atype` type can be used to generate objects representing instances of the signal `a`.

On the other hand firstly more complicated formal constraints can not be represented in a similar simple way, because C++ forbids template parameters that are functions of profile `a* f (atype)` and even of profile `void f (atype)`, where `a` and `atype` are again template parameters.

Second, the use of a variable as a formal context parameter cannot also be captured by this approach. This is because, variables are naturally represented by data members of a C++ class, and data members cannot be template parameters.

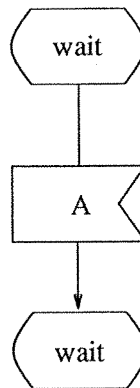


Figure 4: A new transition

Next we discuss representation of inheritance. If an SDL unit $U1$ is defined so that it inherits an SDL unit U, its C++ representation does so just in the same way:

```
class U1 : public U {
  ...                    // classes and pointers for new entities added to U
}; // U1
```

In order to manage addition of a new transition leading from an (old) STATE node, a virtual extension of the STATE node _go method is in advance written in the original service U. This extension is a virtual method with suffix _go_cont1 and has originally an empty body, i.e. a stub. Its call appears within the last statement of the STATE node _go method body and so is operational if none of the previous `if` statements match the signal consumed. See, for example, the implementation of the node method `wait_go` in Example 5 above. After a new transition from a STATE node, triggered by a signal a, appears in the derived service $U1$, the empty implementation of the _go_cont1 method is overridden by the implementation which in effect extends the STATE node _go method to manage the signal a. Let, for example, the new transition exhibited in Figure 4 is added to the service S shown in Figure 3. Then, the derived service is represented by the C++ class S1 given in Example 6 below. Note that the new virtual-extension method `wait_go_cont2` is added into S1 to help further additions.

```
class S1 : public S {
  virtual void wait_go_cont2 ( Signal * s ) {};
  void wait_go_cont1 ( Signal * s ) {
    if ( s -> idcode == A.idcode() ) { next_node = 2; }
    wait_go_cont2 ( s );
  };
}; // S1
```

A newly added transition may contain new nodes. For each of them the corresponding couple of node methods _ask and _go are written down inside the derived service class.

In order to manage the control flow graph of the derived service, we introduce virtual extensions of the switches which implement the ask and go methods of the original service. Those virtual extensions are ask_cont1 and go_cont1. They are used in a similar way as the virtual extension of a STATE node method _go.

SDL virtuals are basically captured by indirect simulation schemas. We hope to explain the relevant details in a posterior paper.

5 Conclusion

We have presented some preliminary considerations involved in compiling SDL 92 into C++ code. In particular we have tried to explain why C++ was chosen as the target code for animation and pointed out the conveniences of this choice in capturing the object oriented features of SDL 92 in a simple manner. We have, on the other hand, also pointed out some of difficulties encountered owing to the differences of SDL 92 and C++ in capturing object orientation. The current project is in progress and we hope to complete the entire compiler design based on the ideas expressed in this paper.

References

[1] CCITT, Functional Specification and Description Language (SDL), CCITT Blue Book, Vol. X - Fasc.X.1, Recommendation Z.100, Geneva (1989).

[2] CCITT, Functional Specification and Description Language (SDL) , CCITT Blue Book, Recommendation Z.100, Geneva (1992).

[3] LOTOS: a Formal Description Technique based on the temporal ordering of observational behaviour, IS 8807 ISO (1989).

[4] P. van Eijk et al., editor, The Formal Description Technique LOTOS, North Holland (1989).

[5] ESTELLE: a Formal Description Technique based on an extended finite state machine model, IS 9074 ISO (1988).

[6] M. D. et al., editor, The Formal Description Technique ESTELLE, North Holland (1989).

[7] Conformance testing methodology and framework, DIS 9646 Part 3: the tree and tabular combined notation (TTCN) ISO, Geneva (1989).

[8] SDT Users Guide and Reference Manual, TeleLOGIC Malmo AB (1992).

[9] Ove Faergemand and Anders Olsen, "New Features in SDL 92", CCITT SDL Newsletter, , 10 (1993).

[10] K. İnan, V. Levin, and H. Yenigun, "Representation of SDL 92 in a CSP kind language, ERPAL", Technical report, SRDC, Scientific and Technical Research Council of Turkey (TUBITAK) (1994).

[11] A. Sezgin. "Comparison of Process Specifications Using the Languages SDL and ERPAL.". Master's thesis, Middle East Technical University, Ankara, Turkey, (1994).

Performance Evaluation

An ATM Based Multiprocessor Communication Architecture 785
M. Salamah, S. Bilgen

Optimising Multivariable Functions Using Tabu Search Algorithm 793
D. Karaboğa, A. Kaplañ

Pattern Recognition

A Continuous Hidden Markov Model with Temporal Correlations 801
S. K. Erdönmez, E. Panayırcı

A Flexible Matching Algorithm for Fingerprint Identification System 807
M. B. Akhan, İ. Emiroğlu

A Second Generation Expert System for Designing Pattern Recognition Applications 815
Y. H. Bugüner, G. Büyükaksoy

Teaching Alphabetical Patterns to Computers Using Neural Network Algorithms and the Effect of Increasing Input Data on Training 823
A. Kalınlı, C. Çiftlikli

ISCIS X The Tenth International Symposium
on Computer and Information Sciences
© 1995 İ.T.Ü. All Rights Reserved

An implementation of SDL 92 communication semantics [1]

Vladimir Levin
Institute of Information Technologies and Electronics (BİLTEN), TÜBİTAK, ODTÜ 06531 Ankara
levin@srdc.metu.edu.tr
Hüsnü Yenigün
Software Research and Development Center, TÜBİTAK, ODTÜ 06531 Ankara
husnu@srdc.metu.edu.tr
Kemal İnan
Electrical & Electronic Engineering Dept, Middle East Technical University, ODTÜ 06531 Ankara
inan@srdc.metu.edu.tr

Abstract. The paper describes an abstract implementation of SDL communication semantics in the larger context of a design of SDL 92 simulation and verification tools. SDL signal exchange involves routing for signal traffic that allows for nondeterminism in communication path selection. The communication path selected may additionally be subject to a path constraint using SDL's *via* construct. We describe first a straightforward decomposition of an SDL program in a CSP like process algebra called ERPAL, and then reduce it to the final ERPAL model by eliminating the no-delay channels, signal routes and gates. The reduction involved simplifies the implementation and yet preserves the intended SDL semantics.
Keywords : Communication Software , Specification Languages, Software Engineering, Compilers

1. Introduction

This paper is a product of ongoing research activity aimed at developing a software platform around the specification language SDL 92 [6]. Currently the main tools under implementation for the SDL 92 platform are a simulator using C++ representation of SDL programs [8] and a finite state verifier [2] involving translation of an SDL subset into COSPAN . Both these tools follow a common scheme to capture the communication semantics of SDL 92. So, prior to specific implementation, the underlying scheme described in the paper has been designed as an abstract model free of details imposed by the implementation languages and the low level implementation considerations. A CSP-like process algebra called ERPAL [3–5] has been designed and selected as an intermediate core structure to express the SDL signal exchange mechanism. Since within the scope of this paper finer aspects of the process algebra language ERPAL are not relevant the reader is welcome to replace the concepts of ERPAL with some extended form of CSP [7] that incorporates variables and guards.
According to the SDL semantics process instances are dynamically created from a process

[1] Research supported by the Scientific and Technical Research Council of Turkey (TÜBİTAK) Project EEEAG COST 247.

definition, thus forming a process instances set referred to as *PiSet* below. The *PiSet* is uniquely identified by the process identifier, whereas a distinct process instance is uniquely identified by its address called *PId*. An SDL program involves a set of communication paths which can consist of delay and no-delay channels, signal routes and gates (see an example on Figure 2(a)). A communication path connects two *PiSets*. Since neither the entire collection of *PiSets* defined in an SDL program, nor their connection by communication paths alter during run time, we abstract from the *PiSet* contents and emphasise the *PiSet*'s role as a placeholder in communication paths. So, we can imagine a *PiSet* to be a box dynamically filled with process instances some of which may eventually cease to exist. The SDL semantics also allows a *PiSet* to be considered a communication mediator which serves routing in the signal exchange of its process instances. In the next section the assembly of communication paths extracted from an SDL program together with the *PiSets*, which they connect, is represented as a graph called the Static Communication Graph (SCG). From a practical viewpoint it is importan that SCG can entirely be constructed by compiler.

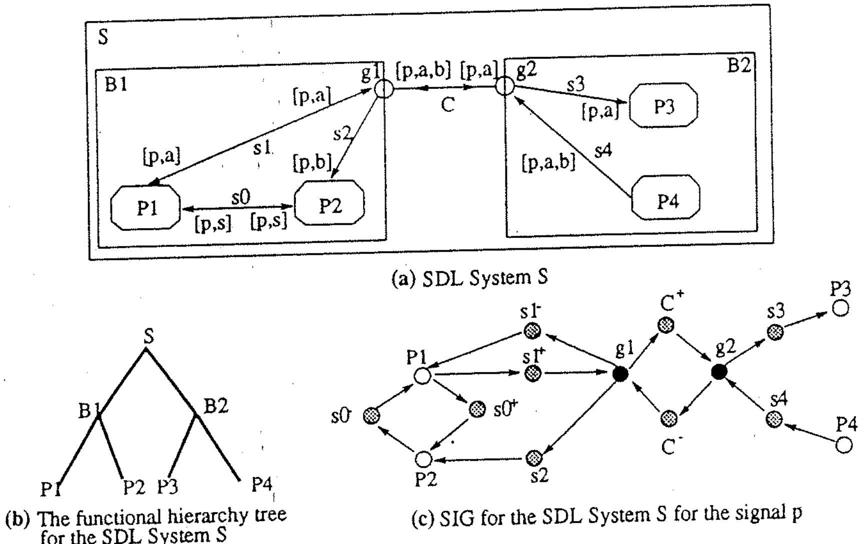
The role of SCG is to be the core data structure for routing a signal sent from an *output* node of a process instance. A signal may be sent to a certain process instance identified by its *PId*, or to a certain *PiSet* identified by its process identifier where the final signal destination is non-deterministically selected among alive processes instances of this *PiSet*, or even to a *PiSet* which itself is non-deterministically selected among such ones which are reachable through appropriate communication paths. An appropriate communication path is that in which all the intermediate links (channels, signal routes, gates) can convey a signal of the given type. An additional constraint is imposed on the communication path, if the *output* node contains a *via* set: the communication path must pass through at least one intermediate link contained in the *via* set. This constraint can be combined with each of the three variants of signal destination.

The above mentioned intrinsic features of the SDL communication semantics are motivated by incremental development methodology for a large distributed software whose components are independently developed by several developers and can change. The software design should provide for stable interfaces for its components. In the SDL context software components are basically blocks encapsulating *PiSets*. Since, *PiSets* of one block are not visible within other blocks, in order to establish communication between *PiSets* located within different blocks, the designer could make use of signal types, channels and gates to define block interfaces and *via* constructs to specialize a wide-spectrum block interface to specific interface cases.

2. Static Communication Graph

We derive the SCG and associated concepts in a number of steps. We start by defining the SDL *static interconnection graph* (SIG^s) of an SDL program for every distinct SDL signal s within the SDL program as a directed graph [2] $G = (V, E)$ where V denotes the vertices and E denotes the directed edges of G which is a subset of $V \times V$. Every gate

[2] In order to keep the notation simple we drop the superscript s in SIG^s and do not allow it to proliferate into the subsequent graph notations. The superscript s simply reminds the reader that the underlying graph is for a single signal s only.



(a) SDL System S

(b) The functional hierarchy tree for the SDL System S

(c) SIG for the SDL System S for the signal p

Figure 1. The functional hierarchy tree and the SIG for an SDL system S

and *PiSet* process and every signal route and channel that carries the signal s in the SDL program is an element of V. If k, n are vertices corresponding to gates [3], and/or *PiSet* processes such that the channel or signal route m connects k directed towards n and carries the signal s in the SDL program graph then $(k, m), (m, n) \in E$. This defines the SIG for each signal which appears in an SDL program. In Figure 2(a) and 2(c) we give an example of an SDL program and its corresponding SIG for the signal p.

In SDL, a communication path connecting two *PiSets* either consists of a single signal route as its intermediate link if the two belong to one block as, for example, $P1$ and $P2$ in figure 2(a), or has an intermediate sequence of signal routes, channels and gates which first climbs along structural hierarchy levels to an upper block or the system, connects two blocks defined within this upper block or the system and then descends along some other branch of the structural hierarchy to the destination process instances set. No block or system level can be entered twice by a communication path. Hence, not every simple path in the graph SIG corresponds to an actual segment of a communication path in the SDL system. For example, neither $P1, s1^+, g1, s2, P2$ nor $P4, s4, g2, C^-, g1, C^+, g2, s3, P3$ is a legitimate communication path. The described above structure of a communication path prohibits purposeless looping of a signal within an SDL system, which further implies that the effective communication graph must be an acyclic graph. The way we obtain this acyclic graph from the SIG is by splitting gate and *PiSet* vertices. The former is split so that gates cannot cause looping and the latter in order to distinguish the message sending and message receiving capacity of SDL processes.

[3] For simplicity we have omitted the SDL 88 residues such as connection points since they can conceptually be replaced by gates.

In order to formally capture the concept of a *legitimate communication path* or its segment we map a path in SIG on the structural hierarchy tree T of the SDL program. At the top of this hierarchy is the entire system represented by the SDL program. Its lower levels are nested blocks and the lowest levels are *PiSets* (see the example in Figure 2(b)). So, every vertex in SIG maps into a vertex in T. Let $h : V \rightarrow H$ be this map, where H denotes the set of vertices of T. We define that $h(v)$ is vertex v itself if it is a *PiSet*; otherwise v maps into the block or system to which it belongs. Now consider any simple path $v1, v2, \ldots, vK$ in SIG and its projection on tree T which is the sequence of vertices $h(v1), h(v2), \ldots, h(vK)$. One obvious SDL-induced feature of the mapping h is that whenever $(u, v) \in V$ in SIG then either $h(u) = h(v)$ or $\{h(u), h(v)\}$ is an edge of the undirected graph T. This simply follows from the simple SDL rule that a channel cannot transgress several nested blocks without being captured by a gate at the boundary of each block.
Now we observe the following two properties of SIG graphs that follow from the construction of SIG from the valid SDL program :
(1) For every gate vertex g of SIG, all the adjacent vertices of g - which are necessarily channel or signal route vertices by construction – can be classified in two disjoint nonempty sets of vertices V_g^+, the *upstream* set and V_g^-, the *downstream* set with the properties that : $v \in V_g^+$ implies that $h(v)$ is a parent of $h(g)$ in T and either $(v, g) \in E$ or $(g, v) \in E$; and $v \in V_g^-$ implies that $h(v) = h(g)$ in T and either $(v, g) \in E$ or $(g, v) \in E$.
(2) For every *PiSet* vertex p of SIG, every adjacent vertex v of p - which are necessarily signal route vertices by rules of SDL - satisfies the rule that $h(v)$ is a parent of $h(p)$ and either $(v, p) \in E$ (p is a sink vertex) or $(g, p) \in E$ (p is a source vertex).
So, the construction based on vertex splitting which yields the Static Communication Graph (SCG) from SIG is formally described as follows:

Algorithm for constructing SCG from SIG
(1) Split every gate type of vertex g of SIG into 2 vertices g_u and g_d and modify the edge interconnections as follows :

(a) For every vertex $v \in V_g^+$ if $(v, g) \in E$ let (v, g_d) be a directed edge of SCG and if $(g, v) \in E$ let (g_u, v) be a directed edge of SCG,
(b) For every vertex $v \in V_g^-$ if $(v, g) \in E$ let (v, g_u) be a directed edge of SCG and if $(g, v) \in E$ let (g_d, v) be a directed edge of SCG,

(2) Split each *PiSet* process vertex p into two vertices p_u (source vertex) and p_d (sink vertex) and let all the incoming edges of p in SIG be the incoming edges of p_d and outgoing edges from p be the outgoing edges of p_u in SCG.

Applying the vertex splitting algorithm above to the SIG given in Figure 2(c) we obtain its corresponding SCG in Figure 3 . The relevant property of a corresponding SCG for a given SIG is that each its simple path is a legitimate segment of a communication path of the source SDL program.
Recall that each SCG corresponding to a specific signal s is distinct. However because these graphs all share the same set of vertices we can superimpose these vertices and obtain (as in the original SDL graph!) a single SCG where each directed edge is labelled by a set of signals. The latter is achieved by merging edges between the same set of vertices

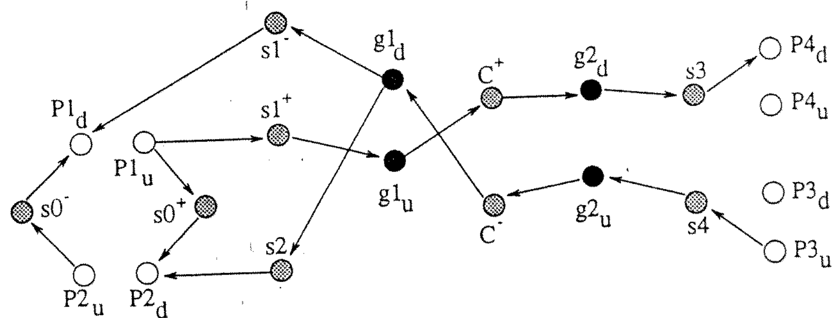


Figure 2. The SCG for the SIG of Figure 2(c)

here on we shall use the term SCG for this global graph where signals are merged within the edges.

Initially, in decomposing an SDL program, we implement each vertex in the SCG of the program as a distinct ERPAL process. During run time this assembly of ERPAL processes are extended to a Dynamic Communication Graph where each *PiSet* process [4] is additionally connected to ERPAL processes modeling its alive process instances [5] . Every SDL channel with arbitrary delay is modelled by an ERPAL process of the buffer type with a single variable of generic queue type to store incoming signals. A no-delay channel or a signal route is modelled by a zero-slot buffer ERPAL process which corresponds to instantaneous synchronization of its input and output actions. Since, such a process is, in fact, a directive to combine two synchronization points these processes corresponding to no-delay channels and signal routes must be removed (see section 3 below). ERPAL processes for a gate and *PiSet* are defined below.
All the ERPAL processes used in the ERPAL model are related to each other via repeated use of a parallel composition operator ||. Although, these ERPAL processes make use of output and input actions which are, in principle, the same as in CSP ($C!m$ and $C?v$ where C is a 'channel tag'), they do this in a specific manner reflecting the SDL communication semantics. Every output action is directed from an active (outputting) vertex o to a single passive (receiving or losing) vertex i. The corresponding input action in i, which synchronizes with the output action in o, may however be also synchronized with output actions of others vertices. In order, to reveal this synchronization aspect of the signal transfer in SCG, we represent the channel tag in the output action directed from o to i as a composed symbol $o.i$ and that in the corresponding input action of i as a composed symbol $f.i$ where o, i are the identifiers of the outputting and inputting vertices regarded

[4]The term *process* denotes by default a process in ERPAL model, whereas the terms *process instance* and *PiSet* are related to the original SDL program.
[5]Each process instance is modelled as a parallel composition of two ERPAL processes: one represents its

[6] and f is a variable whose values are the identifiers of all the vertices in SCG which may output into i.
In formulas below we make use of the following extension of CSP general choice operator:

$$x \in X_1 \times \ldots X_n : [G(x)]A(x) \rightarrow P(x)$$

where x is a list of variables $x_1, \ldots, x_n$ used as parameters in the guard G, event action A and process P, and X_i is the domain of variable x_i. For each value list $x0$ of variables x the boolean expression $G(x0)$ evaluates and if it comes *true* the event $A(x0)$ becomes possible. So, running throughout $X_1 \times \ldots X_n$, variables x induce the set of possible events $\{A(x) : x \in C|G(x)\}$; one of them, let $A(x0)$, is non-deterministically selected to execute and its execution means the transition to the process $P(x0)$.

In order to complete the picture of the ERPAL model based on SCG what remains to be explained is the gate and *PiSet* vertex processes. It is through these processes that we are able to implement the SDL signal routing semantics. Before we list possible outputs we need to be explicit on the exact definition of the ERPAL message parameters which are derived from the SDL signals. The ERPAL message parameters in question consist of a signal identifier s, a *via* set M that may consist of channels, signal routes and gates, a destination process (*PiSet*) identifier d, a *PId* of the destination process instance $r1$, a *PId* of the sending process instance $r2$ and values of the signal parameters $r3$. In formulas below the last three fields are collectively denoted by r. Note that each of fields M, d, $r1$, $r3$ may be empty and then it is denoted by $\emptyset$.
The recursive structure of a gate process c is as follows :

$$Gate = (f.c?(s,d,M,r) \rightarrow \quad ((n \in N_c : [guard(n)]c.n!(s,d,M,r) \rightarrow Gate) \\ |(n \in N_c : [guard'(n)]c.n!(s,d,\emptyset,r) \rightarrow Gate))$$

where N_c is the subset of vertices in SCG each being an outgoing directed neighbour of the gate vertex c. This definition means that after receiving an input message (s,d,M,r) from a vertex f, the gate process c can forward it to a vertex n with the output action $c.n!(\ldots)$ provided that at least one of the two corresponding guards $[guard(n)]$ or $[guard'(n)]$ evaluates to *true*. The former guard serves the situation in which the *via* constraint is not satisfied yet. In this case M is passed to the output action without changing. The latter guard evaluates to *true* if the *via* constraint is satisfied already, or empty, or satisfied just at the gate vertex c. In each of the three cases M is set to the empty set $\emptyset$ in the corresponding output action. The guards are determined at the compile time making use of the SCG. For the sake of simplicity we have omitted the signal discarding action in the *Gate* process when all the guards evaluate to false.
Below we define the guards structures. We assume that each gate or *PiSet* vertex c is equipped with the following two relations :

$$R_{1c} \subseteq S \times V \times V$$
$$R_2 \subseteq S \times U \times \mathcal{P}$$

[6]In the context of SCG and $RSCG$ below we do not distinguish a vertex, the corresponding SDL entity and its identifier - this is possible because each of the three is a static thing.

where S denotes the set of all signals, V denotes the vertices of SCG, $\mathcal{P}$ denotes [illegible] subset of all *PiSet* vertices of V and $U = V \setminus \mathcal{P}$. Note by the use of indices that R[illegible] independent of the local vertex c whereas R_{1g} is not.
If $(s,u,v) \in R_{1c}$ it is understood by definition that there exists a path in SCG [illegible] starts at the current vertex c, moves to the adjacent vertex u and eventually termin[illegible] at v that carries the signal s; u may in particular coincide with v. On the other [illegible] if $(s,u,v) \in R_2$ it is understood by definition that there is a path in SCG starting [illegible] u and terminating at the *PiSet* vertex v that carries s. Below $R_{1c}(s,u,v)$ stands fo[illegible] boolean function induced from the relation R_{1c} and similarly for R_2.
Now assume that the input message (s,d,M,r) demands that a signal s is to be [illegible] a process instance contained in a *PiSet* box with address d which is only parametri[illegible] available at compile time, subject to a *via* path constraint in terms of some $M \subseteq U$[illegible] corresponding guards are then given by the following boolean expressions :

$$guard(n) \quad = M \neq \emptyset \wedge \vee_{m \in M}(R_{1c}(s,n,m) \wedge R_2(s,m,d))$$
$$guard'(n) \quad = (M = \emptyset \vee c \in M) \wedge R_{1c}(s,n,d)$$

In plain language the first guard above evaluates to *true* iff there is a path fro[illegible] adjacent vertex n that carries the signal s, terminates at the destination vertex [illegible] passes through at least one vertex within the *via* set M. The second guard evaluate[illegible] *true* iff the *via* constraint M is satisfied as explained above and there is a path from [illegible] adjacent vertex n that carries the signal s to the destination vertex d.
The structure we have described at the gate processes can be extended to hold a[illegible] process substructure at *PiSet* processes whenever an signal output action is execut[illegible] The extension covers non-deterministic choice of the destination *PiSet* d when it is abs[illegible] in the signal output action. In order, to cope with the non-deterministic choice in questi[illegible] we make use of the set $N_c \times \mathcal{P}$, where N_c is the set of neighbouring vertices of the send[illegible] *PiSet* vertex c, as the set involved in the choice. So, the recursive structure of the par[illegible] the *PiSet* process which is responsible for outputting a signal is as follows :

$$PiSet = \quad (f.c?(s,d0,M,r) \rightarrow \\ (((n,d) \in N_c \times \mathcal{P} : [guard(n,d)]c.n!(s,d,M,r) \rightarrow PiSet) \\ |((n,d) \in N_c \times \mathcal{P} : [guard'(n,d)]c.n!(s,d,\emptyset,r) \rightarrow PiSet))$$

where $d0$ is either the destination *PiSet* identifier or $\emptyset$ stood for empty destination; guards are defined as follows:

$$guard(n,d) \quad = (d0 = \emptyset \vee d0 = d) \wedge M \neq \emptyset \wedge \vee_{m \in M} R_{1c}(s,n,m) \wedge R_2(s,m,d))$$
$$guard'(n,d) \quad = (d0 = \emptyset \vee d0 = d) \wedge M = \emptyset \wedge R_{1c}(s,n,d)$$

3. The Reduced Static Communication Graph

According to the SDL semantics signal routes and no-delay channels should respon[illegible] inputs by immediate outputs with no (delay) interleaving in between. A gate is consid[illegible] simply an interface point of a block where some external channels connected to the bl[illegible] meet some its internal channels or signal routes. Therefore a no-delay channel or sig[illegible] [illegible] implemented as zero-slot buffer ERPAL process or a gate implemented as desri[illegible]

in the previous section gives rise to additional and incorrect interleaving. This is why we remove the no-delay communication links (no-delay channels, signal routes and gates) in the final ERPAL model presented below. The resulting communication graph is called the *Reduced Static Communication Graph* ($RSCG$).

Let $W \subseteq V$ be the set of vertices of SCG that correspond to $PiSet$ and channel vertices with delay. We define $RSCG^s$ corresponding to a signal s as the directed acyclic graph (W, E') where for $u, v \in W$, $(u,v) \in E'$ iff there is a directed path from u to v in $SCG^s = (V, E)$ consisting only of vertices in $V \setminus W$ except for the first vertex u and the last vertex v.

The remaining problem is to extend the solution explained for the SCG to the $RSCG$ where the main obstacle is converting the original *via* constraints involving vertices in $V \setminus W$ into equivalent (in SDL sense) path constraints in $RSCG$. In order, to tackle this problem, we first construct the binary relation $E^s_z \subseteq W \times W$ for each communication link $z \in M$ where M is a *via* set of a signal s. We define E^s_z by letting $(u,v) \in E^s_z$ iff either $z = u = v$ or there is a directed path from u to v in SCG^s which contains z as one of its intermediate vertices and consists only of vertices in $V \setminus W$ except for the first vertex u and the last vertex v. Note that $E^s_z \setminus \{(u,u) | u \in W\} \subseteq E'$.

Next, we replace the *via* set M in the signal structure by the binary relation $E^M = \bigcup_{z \in M} E^s_z$.

Because the gates are removed from the $RSCG$, we redefine each delay channel c to extend its original buffer function with a *fork* function implemented by a separate ERPAL process. The fork process differs from the gate process defined in the section 2 above in the guards only, which, in turn, are re-defined as follows:

$$guard(n) = E^M \neq \emptyset \wedge \vee_{(m,k) \in E^M} (R_{1c}(s,n,m) \wedge R_{1m}(s,k,d))$$
$$guard'(n) = (E^M = \emptyset \vee (c,c) \in E^M \vee (c,n) \in E^M) \wedge R_{1c}(s,n,d)$$

Similar consideration holds for the defintion of $PiSet$ vertex processes.

REFERENCES

1. Bounimova E., İnan K., Levin V., "A Refinement Technique in SDL design of Telephone Exchange", submitted to ISCIS 10, 1995.
2. İnan K., Başbuğoğlu O. "Compiling SDL into the Finite State Specification Language COSPAN", submitted to ISCIS 10, 1995.
3. Yenigün H., "A New Specification Language ERPAL and Translation of SDL-88 into ERPAL", Master's Thesis, Middle East Technical University, Ankara, Turkey, 1995.
4. Sezgin A., "A Comparison of Process Specifications Using The Languages SDL and ERPAL", Master's Thesis, Middle East Technical University, Ankara, Turkey, 1994.
5. İnan K., Varaiya P.P., "Algebras of Discrete Event Models", Proceeding of the IEEE, Vol. 77, No. 1, January 1989.
6. "Functional Specification and Description Language (SDL)", CCITT Blue Book, Recommendation Z.100, Geneva, 1992.
7. Hoare C.A.R., "Communicating Sequential Processes", Prentice-Hall International, London, 1984.
8. İnan K., et al., "Concepts for a C++ Animator Design for SDL 92", In: V.A.Nepomniaschy (ed.), Specification, Verification and Net Models of Concurrent Systems, Institute of Informatics Systems of Siberian Branch of Russian Academy of Science, Novosibirsk, 1995.

ISCIS X The Tenth International Symposium
on Computer and Information Sciences

© 1995 İ.T.Ü. All Rights Reserved

RapPro: Software Rapid Prototyping CASE Tools and Environment

Wie Ming LIM

Temasek Polytechnic of Singapore, School of IT and Applied Science
21 Tampines Ave. 1, Singapore (529757)
e-mail : wieming@tp.ac.sg
Phone : (65) 4704232; Fax : (65) 4756763

Abstract Object-Oriented System Analysis methodologies have being used as r prototyping approaches to overcome the problem faced by many traditional system ana methods of not being able to anticipate and accommodate changes in users' requirements. means and products of such approach are usually object models. These models, however abstract representation of the system with respect to the real world. They are not the 'a system that will be developed and used by the users. We realised that users' requiren changed as they gained 'experience' with the system. The changes needed to accommoda evolved users' requirements will have to be made to the operational systems. This will a the operational status of the organization. To alleviate the problem, we need to push for the users' experience process to the system development phase so that users can learn experience about their system before the real system is being developed. This paper desc a Prototyping Project Model that anticipates and accommodates changes in requirements by allowing them to gain 'experience' with the working models before the a system is being developed. The paper also describes a framework, environment and C Tools that facilitate and support this Prototyping Project Model.

1. Introduction

Object-orientation is believe to be a promising approach towards increasing productivity of system development and its maintainability [Halladay 93]. Object-Ori System Analysis methodologies have being used as rapid prototyping approache overcome the problem faced by many traditional system analysis methods of not being al anticipate and accommodate changes in user requirements. The means and products of approach were usually object models [Connell 95]. These models, however, are ab representation of the system with respect to the real world. They are not the 'actual' sy that will be developed and used by the users in operational environment. Through report experiences we realised that users' requirements changed as they gained 'experience' wit system [Vonk 90]. Even though object-oriented approach and technique facilitate sy evolution and maintenance [Meyer 88], the changes will have to be made to the operat systems. This will affect the operational status of the organization. To alleviate the pro we need to push forward the users' experience process to the system development pha that users can learn and experience with the system. What needed is a 'working' mode conceptually and physically resembling the actual system. Only through such working m users can have real experience with the system and being able to realize the accuracy o user requirements that they had provided initially and therefore modification can be

References

[1] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[2] K. İnan. An Algebraic Approach to Supervisory Control. *Mathematics of Control, Signals and Systems MCSS.*

[3] K. İnan and Pravin P. Varaiya. Finitely Recursive Process Models for Discrete Event Systems. *IEEE Transactions on Automatic Control*, 33(7):626–639, 1988.

[4] K. İnan and Pravin P. Varaiya. Algebras of Discrete Event Models. *Proceeding of the IEEE*, 77(1):626–639, January 1989.

[5] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 1. Technical Report ECS-LFCS-89-85, LFCS, University of Edinburgh, 1989.

[6] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 2. Technical Report ECS-LFCS-89-86, LFCS, University of Edinburgh, 1989.

[7] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[8] A. Sezgin. A comparison of specifications using the languages sdl and erpal. Master's thesis, Middle East Technical University, 1994.

[9] Oralp Tekuzman. FRP: An Implementation Language For Event Driven Systems. Master's thesis, Middle East Technical University, August 1992.

[10] Peter H.J. van Eijk, Chris A. Vissers, and Michel Diaz, editors. *The Formal Description Technique LOTOS.* Elsevier Science Publishers, 1989.

K7

ISCIS X The Tenth International Symposium
on Computer and Information Sciences
© 1995 İ.T.Ü. All Rights Reserved

COMPILING SDL INTO THE FINITE STATE SPECIFICATION LANGUAGE COSPAN[1]

Osman Başbuğoğlu
Military Electronics Industries
(ASELSAN)
e-mail : basbugog@srdc.metu.edu.tr

Kemal İnan
Electrical & Electronic Engineering Department
Middle East Technical University
e-mail : inan@srdc.metu.edu.tr

Abstract: The paper presents the design of a compiler that converts SDL [1] programs into another specification language COSPAN [2]. Whereas SDL is a continuously evolving ITU-T standard language with a Turing machine complexity used by telecommunication industries for system description and embedded software implementation, COSPAN is a finite state machine (FSM) based specification language used both for hardware and software design with powerful verification tools. The practical motivation for implementing an SDL-COSPAN interface is to make use of the COSPAN verification tools. The paper describes the interface design which is currently being implemented after a brief introduction into the SDL and COSPAN languages.

Keywords: Formal methods, specification language, finite state verification.

1. Introduction

SDL is a specification language with reasonably wide industrial usage, especially in communication industries. It is a ISO-CCITT (now ITU-T) standard which evolved with the changing technologies and needs. The most recent version of the standard called SDL-92 was issued in early 1993 with object oriented and other new features. Notwithstanding its industrial usage, it is a well-accepted fact that SDL is not a suitable language for verification applications. First SDL is based on EFSM model and thus it lacks the algebraic structure: powerful asset for both specification and symbolic (proof) verification. Second it makes use of infinite buffers for communication and therefore, in general, the problem whether it can be reduced to an FSM remains undecidable and even if the answer is affirmative the reduction can be resource consuming unless handled with care.

COSPAN is an FSM based specification language developed at AT&T. The verification tools associated with COSPAN are reported to be state-of-the-art (in terms of complexity relief techniques used) in the realm of finite state verification. Recently a timing semantics [3] and associated verifier [4] was added into the verification tools of COSPAN. All this is enough reason to built an interface from SDL into COSPAN since the former does not

have any algebraic structure for using computer aided theorem proving (c.f. LOTOS [5]) for verification and trying to reduce SDL into an FSM seems one way out. Judged in this perspective the present paper can be viewed as a systematic approach for reducing a subset of SDL into a special finite state structure -i.e. COSPAN- particularly suitable for verification.

In the next two sections we present brief overviews of the languages SDL and COSPAN respectively. Section 4 describes the semantics of SDL-COSPAN conversion in the context of the protocol classic : the alternating bit protocol.

2. An overview of SDL

SDL is a graphics based language used for distributed software specification and design. The structural aspects of the software is captured by nested blocks where each block specifies the interconnection architecture of the components of the software at appropriate levels of abstraction. The leaves of the nested tree structure of blocks are called SDL processes. SDL processes are the dynamic EFSM (Extended Finite State Machine) based components of the system that communicate with each other through structures called channels which are modeled by infinite FIFO buffers. Signal routes are special channels connected to processes with instantaneous communication occurring between end points.

Every SDL process is structured in a way that responds to inputs read from its input (message) queue by issuing outputs to various external connections and/or making internal assignments to its variables. Input port of a process is defined (by the standard) as an infinite queue that receives inputs from external connections.

SDL processes violate the simplistic EFSM structure in at least two ways. The first violation arises dynamic creation of procedure instances. The second is due to recursive use of SDL procedures. An SDL procedure is just like a separate process that is instantiated within an SDL process and uses the same input port of the calling SDL process.

In addition to the exceptions to EFSM structure described above the infinite buffer structure of input ports and communication channels are additional aspects that may give rise to infinite state specifications.

Since the target language COSPAN is a finite state one it is natural that we select a subset of SDL that does not allow infinite state specifications. Hence we do not allow constructs like dynamic process spawning, or recursive SDL procedure calls and similar structures that violate the EFSM model. In addition we restrict input ports and communication channels to finite slot buffers. The exact size of these buffers is interactively managed together with the verification steps. More precisely, after assuming a finite size for all buffers and imposing a blocking condition for flow control not to lose data in synchronization the verification exercise is performed on the finite state model in COSPAN. If the verification fails because of the restricted buffer size at some input port or channel, then the buffer sizes are relaxed (enlarged) and the verification step is repeated[2].

There are various destination specification formats for output events. If nothing is specified, the signal is delivered to an arbitrary process instance that is reachable through channels and signal routes and has the signal in its input set. If the PID (Process Identification) of the destination is specified by *to* clause, the signal is delivered to this specific instance of the destination process. It is also possible to constraint the set of communication paths (signal routes and channels) using the *via* clause. In all these formats, when no destination process satisfying the conditions specified in the clauses is found, the signal is simply discarded. If there is more than one process satisfying the conditions, one is chosen non-deterministically.

Timers are set explicitly in SDL processes. When a timer expires it inserts a timeout si[illegible] to the input port of the process that set it. When a timer is the timeout signal already [illegible] into the input port of the process is removed.

3. An overview of COSPAN

In COSPAN, systems and the properties to be verified are both modeled as COSPA[illegible] processes which are based on FSMs. In COSPAN a system consists of a set of concu[illegible] executing processes. In COSPAN both the system and the property it is supposed to [illegible] are modeled as processes. In order to capture liveness (eventuality) property, COSP[illegible] processes are further endowed with additional cycle and edge specifications that mo[illegible] their ω-language behavior [ω-automata]. The entire system is modeled as the produc[illegible] component COSPAN processes. If S stands for the system, and P for the process th[illegible] represents the property to be verified, the entire verification exercise is to compute t[illegible] language containment condition $L(S) \subseteq L(P)$, which in turn can be conveniently ex[illegible] as the ω-emptiness condition of the language $L(SxP^c)$ where P^c is the logical comp[illegible] of the property P. This is the principal logic behind the COSPAN verifier.

In COSPAN each process has selection and state variables. The state variables are [illegible] model the variables in the usual sense and selection variables model the non-determi[illegible] choices made during the execution of a process.

Transitions of the processes are labeled by Boolean predicates built on selection vari[illegible] The transition is made if the predicate in the label evaluates to *true*. The state transiti[illegible] processes consists of two phases: Selection phase and resolution phase. In the selecti[illegible] phase, in all processes of the system, a value for the selection variable is chosen. Thi[illegible] made by the assignments to selection variables. In the resolution phase, the predicate[illegible] processes are evaluated based on to the selections of all the processes, and each pro[illegible] makes a corresponding transition. After each transition, state variables are assigned n[illegible] values.

COSPAN is a synchronous model. During the verification process, COSPAN proces[illegible] simultaneously proceed one step. The synchronization between processes is achieved [illegible] predicates. A process may check the value of the variables of the other processes an[illegible] action accordingly by means of the predicate built on these variables. On the other h[illegible] SDL processes send and receive signals. So, to simulate the communication mechani[illegible] SDL in COSPAN one has to imitate this asynchronous characteristics of SDL in COSPAN's synchronization scheme.

4. Description of the conversion process

In this section we shall describe the conversion of SDL specifications to finite state structures throughout we illustrate our approach on the alternating bit (AB) protoco[illegible]

The SDL diagram for the AB protocol is given in Figure 1. The extra process Chann[illegible] used to simulate the imperfect behavior of a non-ideal channel.

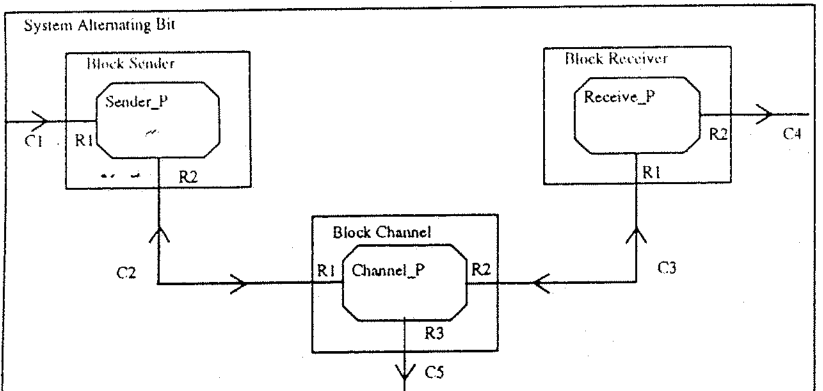


Figure 1 SDL program for the alternating bit protocol specification.

In Figure 1, there are four blocks: system block, sender block, channel block and receiver block. C1, C4 and C5 are channels connecting the blocks to the environment and C2 and C3 connect the blocks to each other. R1, R2 and R3's denote the signal routes within each block.

In Figure 2 we illustrate the corresponding COSPAN process structure. These figures illustrate the process decomposition rules for COSPAN conversion: every SDL process body (3 such processes, Sender_P, Channel_P and Receiver_P) every input port of an SDL process (processes with BuffEnt suffix in Figure 2), every channel (but not signal routes) (processes C21, C22, C31 and C32 in Figure 2) and every timer corresponds to a COSPAN process. As seen from Figure 2, the decomposition consists of 11 such COSPAN processes where arrows depict the synchronization patterns between them.

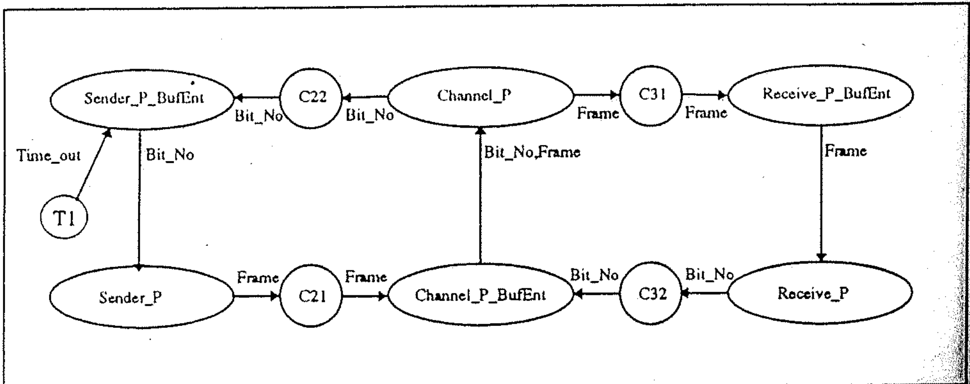


Figure 2. COSPAN processes of the SDL specification

Basically there are 3 kinds of COSPAN processes that correspond to SDL entities. these are:

1. Processes that correspond to input port and channels.
2. Processes that correspond to timers
3. Processes that correspond to SDL process bodies

In what follows we describe type (1) and type (3) processes. Since type (2) processes are straightforward we omit them.

4.1 Input port and channel processes in COSPAN

There are, in essence, finite input blocking FIFO buffer processes where blocking ensures flow control.

The state variables of these processes are the SDL signal types together with signal parameters. Additional variables associated with a signal are the addressing variables th[at] hold the origin-of-message, destination address and path constraint (if any). These additional information are required for the implementation of implicit routing semanti[cs of] SDL communication [1].

In order to illustrate an input port process we present below the input port of the rece[iver] process. From the additional comments following the coded lines it can be observe[d that] this is a simpler one slot buffer that outputs the full buffer or receives an input (at idl[e]) only when the buffer is empty.

```
proc Receiver_P_BufEnt                              /* Buffer entry process of Receiver_P */
stvar $:(idle,Frame_t)                              /* Idle and the type of the signals are the states */
selvar #:(deliver_Frame,receive_Frame_from_C31)     /* Non-deterministic choices of the process */
stvar Frame_vars:Frame                              /* Holds the value of the signal Frame*/
/*Assign the value of the variable Frame_var of the process C31 (channel 31) if the selection of this process is */
/*receiving from the channel and if the channel wants to deliver the message to this process */
asgn Frame_vars -> (C31.Frame_var) ? ((#=receive_Frame_from_C31)*
                        (C31.#=deliver_Frame_to_Receiver_P_BufEnt)) | Frame_vars
init $:=idle,Frame_vars:=0                          /* Initialization of the state and the signal value */
trans                                               /*State variable represents the signal type and existance */
  idle    {receive_Frame_from_C31}                  /* The only possibility is to receive if the state is idle */
    -> Frame_t  : (#=receive_Frame_from_C31)*       /* Change state to Frame_t if synchronization is acheived */
                 (C31.#=deliver_Frame_to_Receiver_P_BufEnt)
    -> $      : else;                               /* Stay at idle state otherwise */
  Frame_t  {deliver_Frame}
    -> idle   : (Receiver_P.#=receive)*(#=deliver_Frame)   /* Back to idle state if the signal is delivered to Receiver_P
    -> $      : else;
end Receiver_P_BufEnt
```

The above code is a typical example of a COSPAN process. State and selection vari[ables] are declared with *stvar* and *selvar* keywords respectively. # is the selection variable [of the] process and its value shows the current selection of the process. The range a selecti[on] variable may take in the curly brackets at each state.

The value of the state variable $ denotes the current state of the process. Other sta[te] variables (e.g. *Frame_vars*) have assignment statements following the keyword *asg[n]*; the assignment to $ is made following the *trans* keyword. Transitions are guarded b[y] predicates (* denotes logical AND operation). At each state the next state and its g[uard are] given after an arrow. For example the transition from *idle* to *Frame_t* occurs if the selection variable of process *C31* is *deliver_Frame to_Receiver_P_BufEnt* and the selection variable's value is *receive_Frame_from_C31*.

process C31 occurs. The synchronization is again detected by the guard of the assignment statement.

A channel process may synchronize for output with another channel process or with an input port process. The following COSPAN code belongs to the alternating bit example and it illustrates the code of a channel process.

```
proc C31                                                         /* Channel process C31 */
stvar $:(idle,Frame_t)                                           /* Idle and signals are the states */
selvar #:(deliver_Frame_to_Receiver_P_BufEnt, receive_Frame_from_Channel_P)   /* Choices of the process */
stvar Frame_vars:Frame                                           /* Holds the value of the signal */
/*Value transfer of signals occurs if SDL body process Channel_P and this process synchronize by their selections */
/* Assign the value of variable t_mes of process Channel_P if synchronization of two processes realizes*/
/* keep the previous value otherwise */
asgn Frame_vars -> (Channel_P.t_mes) ? ((#=receive_Frame_from_Channel_P)*
                   (Channel_P.#=send_t_mes_to_C31)) | Frame_vars
init $:=idle,Frame_vars:=0                                       /* State and signal value initilaization */
/*State variable represents the signal type and existance */
trans
  idle   {receive_Frame_from_Channel_P}                          /* If idle should accept signal */
    -> Frame_t :(#=receive_Frame_from_Channel_P)*(Channel_P.#=send_t_mes_to_C31)
    -> $     : else;
  Frame_t  {deliver_Frame_to_Receiver_P_BufEnt}                  /* If has a signal should try to deliver */
    -> idle  : (#=deliver_Frame_to_Receiver_P_BufEnt)*(Receiver_P_BufEnt.#=receive_Frame_from_C31)
    -> $     : else;
end C31
```

The process in the example transfers the signal *Frame* from process *Channel_P* to process *Receiver_P_BufEnt*. The value of the signal is transferred via the state variable *Frame_vars*. The assignment *t_mes* variable of the process *Channel_P* to *Frame_vars* is made when synchronization with *Channel_P* process is established.

4.2 Structure of COSPAN counterpart of an SDL body process

The COSPAN counterpart of an SDL body process performs the transitions of the SDL process and preserves the signal values correctly. The state variables and the selection variable of the COSPAN process must provide the correct transitions and the correct synchronization with other processes. This is achieved by defining a state variable for each signal and internal variable of the SDL body process.

In the subset of SDL being implemented, there are five different types of transitions, namely, *output event*, *input event*, *internal transition* (used for assignment and decision) *timer setting* and *timer resetting*. Before and after each transition there will be a COSPAN state. The following COSPAN code segments taken from the SDL body process *Sender_P* of AB protocol illustrate the conversion of these transitions.

The first code segment below performs an output event. The synchronization of two processes is detected by using a guard which has two terms. Each term controls the selection of a process. The first term of the guard used below, checks the selection of process C21 and the second term controls that of the process the code segment belongs.

```
4   {send_packet_to_C21}  /*Send signal*/
  -> 5  : (C21.#=receive_Frame_from_Sender_P)*(#=send_packet_to_C21)
  -> $  : else;
```

In the following segment the input event of a signal and a timeout is illustrated. The guards of the transitions detect either a timeout or the delivery of the expected signal. The synchronization is between an SDL process body and its input port process.

```
6   {receive}           /*Receive either time_out or ack*/
  -> 7  : (Sender_P_BufEnt.#=deliver_Bit_No)                         /* Expected signal received */
  -> 4  : (Sender_P_BufEnt.#=deliver_time_out_Sender_P_T1)           /* Timeout happened */
  -> $  : else;
```

The assignment statements of state variables are synchronized with the transitions of the processes. For the assignment and decision the transition structure of the process passes from a dummy state to provide the synchronization.

```
3   {move}              /*Alternate the bit, make the assginment */
  -> 4  : #=move
  -> $  : else;
```

This is synchronized with the assignment construct.

```
asgn nfts ->  0 ? ($=3)*(#=move)*(nfts=1) |
              1 ? ($=3)*(#=move)*(nfts=0) |
              nfts
```

Which means 0 is assigned to *nfts* if the following predicate (($=3)*(#=move)*(nfts=1)) evaluates true or 1 is assigned if ($=3)*(#=move)*(nfts=0) evaluates true and the current value is kept otherwise.

Setting and resetting a timer is synchronized with the timer process, where in the following code segment both actions are illustrated.

```
5   {set_timer_Sender_P_T1}          /*Set timer*/
  -> 6  : (Sender_P_T1.#=get_set)
  -> 5  : else;
  :
8   {move,reset_timer_Sender_P_T1}   /* Reset timer */
  -> 0  : (Sender_P_T1.#=get_reset)
  -> $  : else;
```

5. Conclusion

We have described a method for converting an SDL specification into a finite state specification described by the language COSPAN. Since SDL is, in general, complex enough to incorporate infinite state specifications, the conversion necessarily uses some sort of truncation. First, it excludes some of the recursively used primitives of SDL that exclusively give rise to infinite state description. Second, it dynamically truncates infinite buffer sizes by finite ones at each step of the verification exercise. The compiler described above is under implementation and once the initial implementation is operational, extensions that incorporate larger subsets of SDL shall follow.

REFERENCES

[1] Functional Specification and Description Language (SDL), CCITT Blue Book, Recommendation Z.100, Geneva 1992.

[2] Z.Har'El, R.P.Kurshan. COSPAN User's Guide. AT&T Bell Laboratories, February 1993.

[3] D.Dill.Timing assumptions and verification of finite-state concurrent systems. *In Proc. Workshop on Computer Aided Verification*, Grenoble, June 1989. Lecture Notes in Computer Science, pages 334-348. Springer-Verlag, 1989.

[4] R.Alur, A.Itai, R.Kurshan, M.Yannakakis. Timing verification by successive approximation, *Proc. 4th Workshop on Computer Aided Verification*, Lecture Notes in Computer Science 663, Springer-Verlag, 1992.

[5] T.Bolognesi, E.Bringsma. Introduction to the ISO specification language LOTOS. *North Holland Computer Networks and ISDN Systems* 14, pages 25-57, 1987.

[6] D.Brand, P.Zafiropulo. On communicating finite state machines. *Journal of the ACM*, vol.30, no.2, pages 323-342, 1988.

ISCIS X The Tenth International Symposium
on Computer and Information Sciences
© 1995 İ.T.Ü. All Rights Reserved

Functional-Logic Programming for Smalltalkers:

The FLOOP system

Zeki O. Bayram
Computer Engineering Department
Bogazici University
Bebek 80815/Istanbul-Turkey
internet: bayram@boun.edu.tr
Fax: 90 212 287 2461
Tel: 90 212 263 1500

Barrett R. Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
internet: bryant@cis.uab.edu
Fax: 1 205 934-5473
Tel: 1 205 934-2213

Hakan Ak
Computer Engineeri
Bogazici Uni
Bebek 80815/Ista

Abstract

Motivated by the goal of being able to manipulate complex objects symbolically, we pro method of integrating functional, logic and object-oriented programming paradigm method assumes the existence of an object-expression evaluator (i.e. the underlying Sm interpreter) and relies on transformations and calls to this object-expression evaluator means of computation. Programs of the combined paradigm consist of conditional rules augmented to incorporate object expressions.

1 Introduction

In this paper, we propose a method of integrating three programming language parad namely *object-oriented, functional* and *logic*, in an intuitive, coherent and practical way advantages of integrating functional and logic programming have been well demons [1,2,3,4,5,6,7,11,14,15]. We are taking those advantages one step further in FLOO incorporating complex objects into the combined functional/logic paradigm. FLOOP system implemented in Smalltalk that achieves the integration of functional, logic and oriented programming paradigms through a functional/logic interpreter base transformations, an independent "object-evaluator" (i.e. the Smalltalk interpreter [19 evaluating object oriented expressions and an interface between the two. A FLOOP pro consists of a set of conditional rewrite rules. The incorporation of the object or

Our current research is directed along two different lines. From the applicative poi[nt] of view, we are developing a software tool that implements the integrated approach [in] the exponential case. From the theoretical point of view, we are addressing the proble[m] of scaling the integrated approach to general distributions. It is however important [to] point out that the limitation to exponentially distributed durations is (i) convenient be[cause] exponential timing allows for a Markovian analysis without resorting to time-costl[y] simulations, and (ii) not so restrictive because many frequently occurring distributio[ns] are (or can be approximated by) phase-type distributions, and these are expressible [in] EMPA by means of the interplay of exponentially timed and immediate actions.

References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, *"A Class of Generalized Stochastic Petri N[ets] for the Performance Evaluation of Multiprocessor Systems"*, ACM Trans. on Compu[ter] Systems 2:143-172, 1984

[2] M. Ajmone Marsan, A. Bianco, L. Ciminiera, R. Sisto, A. Valenzano, *"A LOTOS [Ex]tension for the Performance Analysis of Distributed Systems"*, IEEE/ACM Trans[.] Networking 2(2):151-164, 1994

[3] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, *"A Note on Reliable Full-Du[plex] Transmission over Half-Duplex Links"*, CACM 12(5):260-261, 1969

[4] M. Bernardo, L. Donatiello, R. Gorrieri, *"Integrating Performance and Functional A[nal]ysis of Concurrent Systems with EMPA"*, Technical Report UBLCS-95-14, Universit[y] Bologna (Italy), 1995, available from `ftp.cs.unibo.it:/pub/TR/UBLCS`

[5] G. Chiola, *"GreatSPN 1.5 Software Architecture"*, Proc. of the 5th Int. Conf. on Model[ling] Techniques and Tools for Computer Performance Evaluation, Elsevier, pp. 121-136, 19[91]

[6] R. Cleaveland, J. Parrow, B. Steffen, *"The Concurrency Workbench: a Semantics-Ba[sed] Tool for the Verification of Concurrent Systems"*, ACM Trans. on Programming [Lan]guages and Systems 15(1):36-72, 1993

[7] N. Götz, U. Herzog, M. Rettelbach, *"Multiprocessor and Distributed System De[sign:] the Integration of Functional Specification and Performance Analysis Using Stoch[astic] Process Algebras"*, Proc. of PERFORMANCE '93, LNCS 729:121-146, 1993

[8] H. Hermanns, V. Mertsiotakis, M. Rettelbach, *"Performance Analysis of Distrib[uted] Systems Using TIPP - A Case Study"*, Proc. of the 10th UK Performance Enginee[ring] Workshop for Computer and Telecommunication Systems, 1994

[9] J. Hillston, *"A Compositional Approach to Performance Modelling"*, Ph.D. Thesis, [Uni]versity of Edinburgh (UK), 1994

[10] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989

[11] M. K. Molloy, *"Performance Analysis using Stochastic Petri Nets"*, IEEE Tran[s.] Computers 31(9):913-917, 1982

[12] N. Nounou, Y. Yemini, *"Algebraic Specification-Based Performance Analysis of Co[mmu]nication Protocols"*, Protocol Specification, Testing and Verification V:541-560, 198[5]

[13] R.A. Sahner, K.S. Trivedi, *"Reliability Modelling Using SHARPE"*, IEEE Trans. o[n]

Computer and Information Sciences

1996 METU All Rights Reserved

An Interface Implementation using a Formal Compiler Specification Method [1]

Vladimir Levin, TÜBİTAK-BİLTEN [2]
Eleonora Bounimova, TÜBİTAK-BİLTEN
Kemal İnan, E&E Eng. Dpt., METU
ODTÜ-Balgat 06531 Ankara
(levin,ella,inan)@srdc.metu.edu.tr

Abstract

The paper presents a new formal compiler specification method that has evolved out of a number of realistic applications including the current application reported in the paper and describes how it is used as a formal method tool for compiler design and implementation problems.

Keywords: Compiler, Formal Specification, Predicate Calculus.

Introduction

[Th]is paper presents a byproduct of our research activity aimed at developing a verifica[ti]on/simulation platform around the specification language SDL 92 [1]. Since this platform [in]volves translations from SDL into various target languages - currently into C++, S/R and Promela [4] - we have designed a methodology addressed at the development of a [co]mpiler back end. The key point of this methodology is a calculus on a growing target [de]rivation tree coupled with a given source derivation (parse) tree. The calculus is used [to] specify a translation mapping from a source language to a target one. So, the paper [pr]esents the formalism and describes one of its applications, namely, the specification of a [tra]nslation from SDL into S/R. The complete set of translation rules for our SDL-to-S/R [co]mpiler is given in [5] and explained in detail in [13]. The compiler has been implemented [us]ing Karlsruhe Coctail [7] as a technological toolkit and the translation rules [5] as the [de]sign document.

[W]hereas the development of a compiler front end is a more-or-less straightforward task [su]pported by well-known tools (such as LEX+YACC), this is not the case at all in de[ve]loping a compiler back end. Here the state-of-art tools (such as Karlsruhe Cocktail) [...] only facilitate the implementation phase, but are of no help in designing a transla[ti]on mapping that constitutes the heart of a compiler development. The novelty of the [me]thod described in the paper, in comparison with some of the well-known specification [for]malisms [6, 2, 9], lies in its capability to handle local-to-global contextual dependen[cies] between the source and the target language derivation trees throughout the growth [of] the target tree. The example given in section 4 demonstrates this capability where a [syn]tactically distributed set of assignments to a variable within the source code (in SDL) [is] mapped into a single assignment statement within the target code (in S/R).

[Th]e formalism presented in this paper has evolved from the translation specification ideas [wh]ich were formulated in [3], then applied to specify a translation from Pascal to IBM-360 [asse]mbler [4] and used in a semi-formal way in the Russian Aerospace industrial software

[1] Research partially supported by TÜBİTAK, Project EEEAG COST 247, NATO-AGARD Project [...3] and TÜBİTAK-BİLTEN.

to 1993. A predicate calculus on nodes of a derivation tree ... specifying static semantics of programming languages. Some of the ideas articulated ... the cited works have independently appeared in [14, 15], in the context of static seman... formulation as opposed to entire translations.

In the next section the translation formalism is formulated. Section 3 contains some of ... enrichments in terms of notation, conventions etc. added to the translation formal... Section 4 starts out with a brief exposure to the languages SDL and S/R and conti... with the detailed description of an example extracted from [5] that incorporates a sim... SDL program translation into S/R code. In section 5 we describe the implementatio... the compiler.

2 The Basic Translation Formalism

In this section we first describe the translation system as a translation specification for... ism and then motivate the reader to the underlying intuitive notions and ideas associ... with this formalism.

Let $G = (V, \Sigma, \Pi, \Theta)$ be a context-free grammar where V and Σ stand for the non-term... and terminal symbols of the grammar respectively, $\Pi \subseteq V \times \{\{V \cup \Sigma\}^* \cup \epsilon\}$ stand... the finite set of context-free production rules with ϵ the empty string and Θ stands fo... start symbol in V. Following [8] we define a *derivation tree* τ over the grammar G... ordered, directed tree. We let N denote the nodes and the vertical precedence re... $<$ denote the directed edges (from child to parent) of the tree. For future use we ... denote the transitive closure of $<$: which is a strict partial order. In this ordered tr... original horizontal strict partial order denoted by $<$ is induced by the left-to-right ... on the children of a parent node. We use the symbol $\overline{<}$ to denote the extension of ... letting $x \overline{<} y$ whenever x and y have ancestors that stand in the relation of being two ... nodes of a common parent where the ancestor of x is smaller than that of y. The ... enjoys the following properties [8] :

(1) There is a tagging function $tag : N \rightarrow V \cup \Sigma \cup \epsilon$ and the *tag* of the roo... node is Θ.
(2) If the children of n are $n_1 < n_2 < \ldots < n_k$ and if $tag(n) = A$ and ... $tag(n_i) = A_i$ then $A \rightarrow A_1 \ldots A_k$ is a production in Π.
(3) If $tag(m) = \epsilon$ - where ϵ also stands for the empty symbol - then m is th... only child of its parent.

An *extended derivation tree* - or an xtree in short - is a derivation tree τ together ... collection $\{f \in \mathcal{F}\}$ of partial (attribute) functions where each $f : N \rightarrow \mathcal{A}$ with $\mathcal{A}$ ... attribute values) and a collection $\mathcal{R}$ of binary relations defined on the node set ... shall use the symbol τ also to denote the xtree. We call an xtree *complete* if all it... have tags in $\Sigma \cup \epsilon$. There is one important derived attribute function that applie... the nodes of an xtree and which is frequently used in the context of matching or ch... the equality of identifiers in compiler specification. Consider any node n in an xt... the subtree of it rooted at this node. Let $l_1 \overline{<} \ldots \overline{<} l_N$ denote the leaves of this ...

[5]Note that the *tag* function can be viewed as a member of $\mathcal{F}$ and the precedence relation ... strict partial relations $<|$ and $\overline{<}$ can also be viewed as members of the set $\mathcal{R}$.

...w we describe the concept of a *translation state* which is used in interpreting the ...slation rules given in terms of special formulas.

...inition 0.1 *Let G_S and G_T denote grammars for the source and target languages ... ...tion. Let τ_S and τ_T denote the corresponding xtrees over the grammars G_S and G_T ...ectively. A* translation state *$\mathcal{S}$ is a triple (τ_S, τ_T, R) where τ_S is a complete xtree and ...esentation relation $R \subseteq N_S \times N_T$ where N_S and N_T denote the nodes of the trees ...nd τ_T respectively. We define a partial order on translation states by letting $\mathcal{S} \sqsubseteq \mathcal{S}'$ ...e $\mathcal{S} = (\tau_S, \tau_T, R)$, $\mathcal{S}' = (\tau'_S, \tau'_T, R')$ iff, first, τ_S and τ_T are sub-xtrees of τ'_S and τ'_T in ...ense that the underlying graphs are subtrees and relations and functions of the xtrees ...nd τ'_T agree with those of τ_S and τ_T on the common nodes, and, second, R' agrees ... R on the common nodes.*

...we define the *translation derivation rules* to be interpreted by the translation states ...anslation derivation rule is written as $\frac{p(x)}{c(x,y)}$. Here $p(x)$ (the *premiss*) and $c(x, y)$ (the ...*clusion*) are formulas where x and y denote the lists of free variables that occur in the ...pective formulas. Each free variable is a node variable that can be interpreted by th... ...des of the xtrees of a translation state $\mathcal{S}$. The formulas mentioned above belong to ... ...order predicate calculus the specific details of which are briefly summarized below

...erpretation domains for formulas are translation states. If $f(z)$ is a formula with ... ...of free variables $z = z_1, \ldots, z_k$, where each z_i is a node variable , $\mathcal{S}$ is a translation ...and $\hat{z} = \hat{z}_1, \ldots, \hat{z}_k$ is a list of nodes in the xtrees of the translation state $\mathcal{S}$ then w... ...e $(\mathcal{S}, \hat{x}) \models f(z)$ whenever the interpretation domain $(\mathcal{S}, \hat{z})$ is a model for the formu... ...that is, the open formula $f(z)$ is interpreted at the translation state $\mathcal{S}$ subject ... ...substitution of the values $\hat{z}_i$ for the variables z_i.

...principal predicates and functions of the predicate calculus in question are presente... ...w where $y, x, x_1 \ldots x_k$ are node variables interpreted by nodes in τ_S or τ_T:

(1) Production predicates of the form $y \rightarrow x_1 \ldots x_k$. This evaluates to true iff in the respective tree (within the translation state) the nodes x_1 to x_k are the ordered children of the node y.
(2) Derivation predicates of the form $y \Rightarrow x_1 \ldots x_k$. This evaluates to true iff in the respective tree the nodes x_1 to x_k are some left-to-right ordered children, grandchildren, grand-grandchildren, etc. of the node y that make up a maximal subset of such nodes (children, grandchildren, etc.) which is yet totally left-to-right ordered. A derivation predicate can be also characterized as follows : the string $tag(x_1) \ldots tag(x_k)$ is a sentential form ([8]) derived from $tag(y)$.
(3) Binary predicates of the form $x \xrightarrow{Rel} y$ where x and y are interpreted by the nodes of xtrees and Rel by any of the relations of the xtrees.
(4) The predicate $x \xrightarrow{R} y$ is interpreted by representation relation R.
(5) Attribute functions of the xtrees.

...ddition to above relations, first, the equality predicate '=' for both nodes and a... ...butes and, second, predicates and operations built in the attribute types making ...

body.

We are now in a position to define a *translation system* and a *translation derivation*.

Definition 0.2 *A translation system is a pair* (S_0, π) *where* S_0 *is an (initial) transla[tion] state and* π *is a finite set of translation rules. We define a* translation derivation step *as follows : assume that* S *is the current translation state then we can derive a transla[tion] state* S' *by invoking a rule* $r = \frac{p(x)}{c(x,y)}$ *provided that for some list of node values* $\hat{x}$ *in* S

1. $(S, \hat{x}) \models p(x)$
2. $S \sqsubseteq S'$
3. $(S', \hat{x}) \models \exists y\, c(x,y)$
4. S' *is a* *minimal* *extension of* S *satisfying conditions 2) and 3) above* – **i.e.**

$$\{S \sqsubseteq S'' \sqsubseteq S' \wedge (S'', \hat{x}) \models \exists y\, c(x,y)\} \implies S'' = S'$$

If S *derives to* S' *as defined above and* $S \neq S'$, *we call this derivation step produ[ctive] and denote it* $S \overset{r}{\vdash} S'$.

Note that the target xtree τ_T at any stage of a translation derivation, by definition of [the] translation state, must be in conformity with the grammar G_T.

A translation derivation denoted $S_0 \overset{\pi}{\Rightarrow} S$ is a short-hand notation for $S_0 \overset{r_1}{\vdash} \ldots$ where each $r_i \in \pi$. A translation state is called *terminating* if no rule applies for a[ny] productive derivation step. It is called *complete* if the target xtree of the final sta[te is] complete. It is called *deadlocked* if it is terminating but not complete.

3 The Enriched Translation Formalism

In this section we enrich the notation and conventions of the basic formalism expla[ined] in the previous section in order to apply it to a realistic translation problem.

In practice the target grammar G_T used in the basic translation formalism may dev[iate] from the original grammar of the target language which can be used, for example [in the] target language reference manual. It is natural and also our own experience tha[t the] translation designer develops G_T in a specific way to suit the translation strategy [and] the source language grammar G_S. So, G_T grows up in parallel with design of transl[ation] rules in such a way that the rules suggest target grammar productions which specify [tar]get language counterparts (images) for the relevant source grammar nonterminals. [We] shall therefore call this grammar a *projective grammar* for the translation exercise. [The] projective grammar, if correctly designed, should generate a sublanguage of the origi[nal] target language. [6]

Next we explain some extended features of the compiler specification formalism tha[t are] used in the example in the next section - as well as in general - to facilitate the pr[...]

[6] The actual picture is more complex since we must also take into account the visibility and [...] rules of the target language.

[...] refer to a n[od]e with a tag X as to a node of sort X. Given a tree t, a *node sort* [i]n t is defined as the set of nodes in t with a tag X. A node sort can be used in [qu]antification to restrict a domain of a quantified node variable.

[...] In the actual translation rules, a nonterminal symbol, say X, possibly with a numeric [su]ffix k is used to form a node variable Xk which by default belongs to node sort X. Th[is as]sumes that in the corresponding rule a predicate $tag(Xi) = X$ is implicitly present.

[...] The notation x^t is a short hand notation for the following : the symbol x is already [a no]de variable and every occurence of x^t stands for a new node variable that satisfies th[e co]nstraints $x \sim x^t$ and $tag(x) = tag(x^t)$.

[...] In a grammar we allow extended productions with iterations B^* and B^+ within th[e R]HS of a rule, which can be thus presented in some of the two following forms:

$$A \rightarrow \ldots B^* \ldots \qquad A \rightarrow \ldots B^+ \ldots$$

[As] usual, B^* stands for arbitrary - possibly zero - number of consecutive B symbols an[d] B^+ for arbitrary non-zero number of such symbols. These iterations enable us to replac[e m]ultilevel recursive structures in a tree by simple one level list of nodes of B sort. This [in t]urn reflected in the capability in the target tree to add on further children to an inte[rnal] node of A sort in a successive translation derivation process. In order to accomplis[h th]is by adding further children at such nodes we use the precedence predicate $A :> B$ [Se]e, for example, the predicate $AsgnAlts :> AsgnAlt$ in the rules (2) and (3) in the nex[t se]ction.

[(a)] In writing production and derivation predicates ($\rightarrow$ and $\Rightarrow$), a terminal symbo[l wr]itten in the `Typewriter` font stands for an anonymous node variable denoting a nod[e w]hose tag is this terminal. The sequence of terminal symbols and node variables whi[ch] the RHS of such a predicate is enbraced by apostrophes (′) .

[(b)] Similarly to a chain of inequalites (for example, $a > b < c < d$) on integers, we allow [a s]hort hand notation $x\, q_1\, y\, q_2\, z$ for formula $x\, q_1\, y \wedge y\, q_2\, z$ where x, y, z are node variable[s a]nd q_1, q_2 are symbols of orders or precedence relations on nodes (i.e. $\lhd$, $\rhd$, $\overline{\lhd}$, $\overline{\rhd}$, <, [...]).

[4] Specification of a Translation from SDL into S/R

[B]elow we first briefly outline the translation strategy and then demonstrate some typica[l f]ragments extracted from the set of translation rules [13, 5].

[A]n SDL program consists of processes which communicate to each other either by sendin[g a] signal via channels or by direct insertion of a signal (message) to the input buffer of th[e de]stination process. The graphical syntax of an SDL process (cf. Figures 1 (a) and 1 (b)) [re]sembles that of a program flow chart. Every geometrical shape in this flow diagram [co]rresponds to a program execution except for the start symbol and the state symbo[ls w]hich are simply positions. The kinds of executions could be reading a signal from th[e i]nput buffer - in which case the signal read is consumed - , sending an output, assigning [v]alues to internal variables, evaluating a decision predicate for branching etc.

[T]he SDL program given in Figure 1(a) and its S/R state transition diagram given i[n Fi]gure 1(c) reveals that for every SDL execution there is a - possibly guarded - transitio[n i]n the corresponding S/R process and S/R states simply correspond to the nodal con[ne]ctions of the execution boxes (or start symbols and SDL states as explained above) i[n]

that at each state a transition must be possible and ...
as illegitimate self-looping cycles in the S/R processes.

Unlike SDL which is an asynchronous language with interleaving semantics, S/R is a syn-
chronous language within which all processes move simultaneously under an imagina-
global clock pulse. The coordination between S/R processes is provided by sharing
lection variables which are used in transition guards (predicates) and transition actio-
(assignments). Figure 1(c) demonstrates a control flow diagram of a distinct S/R pro-
cess where *Input__Sgn*, *Input__Par* are selection variables. A special selection variable
(hidden in Figure 1(c)) is used to model the selection among possible transitions : it
non-deterministicaly assigned one of the values listed at a state position and then is u-
in the transition guard. For example, in the S/R process exhibited by Figure 1(c), th
values of # are *Consume* , *Wait* , *Move* , they are used in the guards *ConsumeCo*
, *MoveCond* which stand, respectively, for predicates # = *Consume* and # = *Mo*
Whereas assignment for state variables may be recursive in nature where previous valu
of the state variable may be used in the assignment, the assignment for selection variab
is not allowed to be recursive; it is simply a read-out function of the state variables an
other selection variables in the absence of recursion. For example, in the S/R proc
exhibited by Figure 1(c), there is only one state variable, x. For further details abo
S/R the reader is refered to [10].

In the S/R image of a source SDL program, there is a distinct S/R process for each SD
process body, for the input buffer of the latter, for each SDL channel and timer. In th
S/R counterpart of an SDL process body, an S/R state position corresponds to each SD
node in the SDL process flow diagram. An SDL variable is modeled in S/R as a sta
variable. The transition structure of the S/R process corresponding to an SDL proce
is directly provided by the flow diagram of the SDL process (see Figures 1(a) and 1(c
In order to capture the asynchronous message passing feature of SDL some general S/
constructs are required. These are selection variables such as *Input__Sgn* , *Input*
and transition guards such as *SendCond* , *ConsumeCond* . Furthermore a finite de
self-loop is added to some S/R state positions to model the asynchronous semant
SDL - see, for example, a self-loop at state position A in Figure 1(c).

As a strategic decision we have discarded all the constructs within SDL that may co
spond to potentially infinite state situations: dynamic process spawning, recursive pro
dure calls etc. On the other hand we have decided to represent the infinite SDL que
used in the input buffers and the channels by finite blocking buffer processes in S/R

We present here four rules from [5] which concentrate around the translation mappi
SDL variables. The mapping is not trivial, since the S/R language only allows a sing
signment for each variable. So, all the assignments to an SDL variable should be coll
into one S/R assignment. To clarify this aspect of both languages and the transla
strategy, we present in Figure 1 a simple example of an SDL process and its image in
language, concentrating on the details of variable mapping. In the explanation whi
follows we only take into account the features of both languages relevant to our extra
rules.

In Figure 1, an example of a simple SDL process (Figures 1(a), 1(b)) and its image

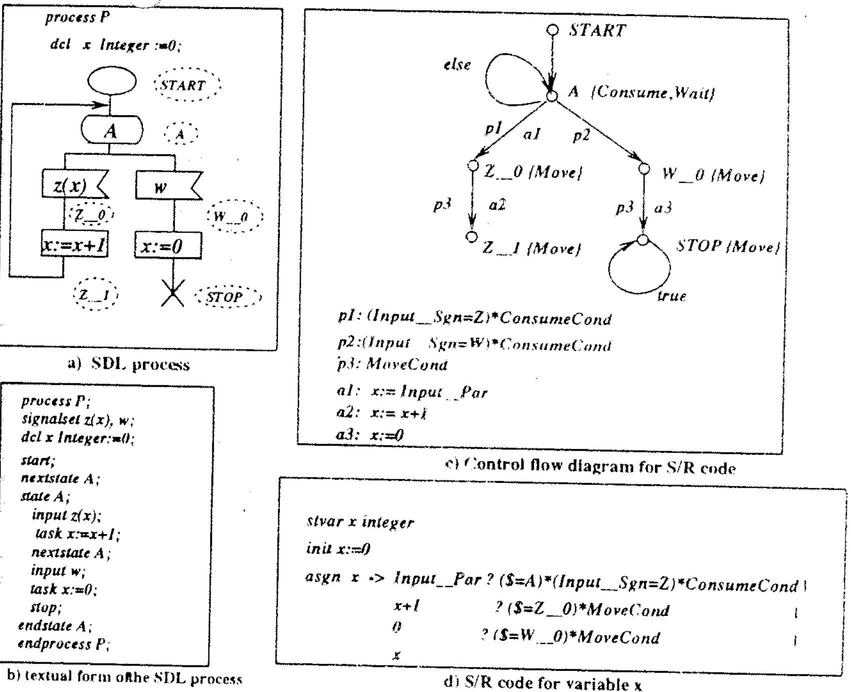


Figure 1:

...ntrol flow diagram for S/R code (Figure 1(c)) are presented. Additionally, S/R code ...r the SDL variable x from the example SDL process P is given in Figure 1(d). An ...DL variable is mapped into an S/R state variable. The latter must be declared as **stvar** ...nd initialized. In S/R, all the initializations are collected into a single statement, which ...arts with the keyword **init**. The third component of S/R code dealing with the state ...ariable is its single assignment statement, which is a block of alternatives. Each of these ...lternatives presents a distinct assignment to this variable which is attached at some ...ansition in the control flow diagram of the S/R process - the body of the alternative ...followed by symbol ? - is simply the RHS of the corresponding assignment, and the ...quel predicate relates the alternative to the corresponding transition in the S/R pro...s control flow diagram. So, in S/R code, all the different locations of assignments to ...distinct variable are collected within this single assignment block with the guards dif...entiating these locations. According to the semantics of SDL - or more precisely the ...bset of SDL chosen for translation – there are two distinct contexts in which values are ...signed to a variable : an SDL assignment and an input construct in which the variable ...used as a parameter of an input signal. In order to refer to these contexts, we include ...to the translation relation R a sub-relation which binds the relevant SDL constructs to ...ate positions in the S/R process control flow diagram. This sub-relation is defined in ...anslation rule (4) and is used in translation rules (2) and (3), see below. In S/R code, ...state position is labeled by a name which can be referred to as a current value of the ...ecial state variable $\$$. These labels - more precisely, their defining occurrences - are

position). In Figure 1(a), the dashed cycles exhibit the labels of the S/R state positio[illegible] to which the corresponding SDL constructs are bound.

For our example (see Figure 1(d)), the assignment for x includes the three alternati[illegible] extracted from the Figure 1(c), the references to the state positions being denoted [illegible] state variable \$. .Guards *ConsumeCond* , *MoveCond* are explained above. The [illegible] alternative of the assignment generates a default value where the value of x does [illegible] change in case the guards of the previous three alternatives evaluate to *false*.

Below four derivation rules for mapping SDL variables are given. [7]

$$ProcDf :> VarDf \;\wedge\; VarDf \rightarrow Var0\ Type\ Expr0 \;\wedge$$
$$VarDf \xrightarrow{R} VarDfIm \;\wedge\; ProcDf \xrightarrow{R} ProcDfIm \;\wedge\; ProcDfIm :> ProcIm :> InitVals$$

$VarDfIm \rightarrow$ 'stvar $VarIm0$: $SRType\ AsgnSt$' $\wedge$
$AsgnSt \rightarrow$ 'asgn $VarIm0^t$ −> $AsgnAlts\ VarIm0^t$' $\wedge$
$InitVals$:>' $VarIm0^t := ExprIm0$' $\wedge$
$Var0 \xrightarrow{R} AsgnAlts \;\wedge\; Expr0 \xrightarrow{R} ExprIm0 \;\wedge\; Type \xrightarrow{R} SRType \;\wedge\; Var0 \sim VarIm0$

S/R Projective Grammar rules:
$AsgnSt ::= VarNm$ '−>' $AsgnAlts\ VarNm^t$
$AsgnAlts ::= \{AsgnAlt\}^*$
$AsgnAlt ::= Expr$ '?' $Expr$ '|'

$$Asgn \rightarrow Var\ Expr \;\wedge\; Var \xrightarrow{DfOc} Var0 \;\wedge\; Var0 \xrightarrow{R} AsgnAlts \;\wedge\; Asgn \xrightarrow{R} SRSt$$

$AsgnAlts :> AsgnAlt \;\wedge\; AsgnAlt \Rightarrow$ '$ExprIm$? (\$ = $SRSt^t$) * MoveCond |' $\wedge$
$Expr \xrightarrow{R} ExprIm$

$St > StNm \;\wedge\; St \rhd Stim :> (SgnId, SgnPar) \;\wedge$
$SgnPar \xrightarrow{DfOc} Var0 \;\wedge\; Var0 \xrightarrow{R} AsgnAlts \;\wedge\; StNm \xrightarrow{R} SRSt$

$AsgnAlts :> AsgnAlt \;\wedge$
$AsgnAlt \Rightarrow$ 'Input_Par ? (\$ = $SRSt^t$) * (Input_Sgn = $SgnIdIm$) * ConsumeCond |' $\wedge$
$SgnIdIm \sim SgnId$

$x \xrightarrow{R} SRSt$

$$SRSt \rightarrow SRStNm \;\wedge\; \forall x1 \in N, s1 \in SRSt(x \neq x1 \wedge x1 \xrightarrow{R} s1 \Longrightarrow SRSt \not\leadsto s1)$$

Rule (1) above specifies the S/R images for an SDL variable which appear in a [illegible] tree (basically as placeholders for the corresponding S/R constructs) in translatin[illegible] variable declaration. Note that this rule in particular refers to the context (*InitVal*[illegible] S/R code where the single (for all the variables) initialization statement has been lo[illegible] On the other hand, the rule creates a new context (*AsgnSt*) in S/R code whe[illegible]

[7] The identifiers in the rules are abbreviated in accordance with the following conventions: *Df* [illegible] for *Definition*, *Proc* - for *Process*, *Im* - for *Image*, *Alt* - for *Alternative*, *Sgn* - for *Signal*, *St* - for *St*[illegible] - for *Name*, *DfOc* - for *Defining Occurrence*, *SR* - for S/R (language), *Stim* - for *Stimulus*.

[illegible]gle) assignement statement for the variable in question will be located.

[illegible] (1) is followed by the projective syntax of an S/R assignment statement [illegible]cifies the assignment alternative (refer to rule (1), the component *AsgnAlts* [illegible] [illegible]gnment) that corresponds to the SDL assignment . In case there are sever[illegible] [illegible]nts for an SDL variable, one alternative is added for each assignment in [illegible] [illegible] rule (2). Rule (3) defines the assignment alternative that corresponds to [illegible] [illegible]ce of the SDL variable as a parameter of an input signal. This rule is very [illegible] [illegible] previous one, only the structure of the assignment alternative is slightly dif[illegible] [illegible]gure 1 (d)). Rule (4) states that there should be no name collision in the [illegible] [illegible] state positions within the target code.

[illegible] Implementation

[illegible]ter illustrating the method by the rules extracted from the complete specif[illegible] [illegible] SDL-into-S/R compiler, we briefly reflect upon various aspects of the me[illegible] [illegible]ed in the context of this project and state some figures and facts on the pro[illegible] [illegible]y inform the reader regarding the nature of our undertaking.

[illegible]he most challenging part of the methodology used is involved in formulating t[illegible] [illegible]tion rules. The starting points for writing the rules are grammars of the so[illegible] [illegible] target language representations and a conceptual mapping of the source [illegible] [illegible]nstructs into the target language. In the context of this example this mapping [illegible] [illegible]ucted recursively by exploiting the structural hierarchy of the SDL language [illegible] [illegible]om the higher level constructs of SDL (system), the image was expressed via th[illegible] [illegible] the lower level constructs (declarations at system level, enclosed blocks, proces[illegible]

[illegible]he most intricate part of the translation specification occurs when there are rem[illegible] [illegible]lations both demanded by the premisses and constructed by the conclusions [illegible] [illegible]he rules. It is interesting and instructive to note that most of the rules have suc[illegible] [illegible]ode relations. For instance in the example given in the previous section all the f[illegible] [illegible]ed incorporate remote node relations. A typical instance of a remote node [illegible] [illegible]fluencing the target tree extension occurs in the example above when dealing w[illegible] [illegible]riable assignments. Whereas assignment in the SDL syntax may take place in [illegible] [illegible]rts of the program, in S/R language all assignments to a state variable sho[illegible] [illegible]ace within a single location of the program. This gives rise to a distributed-t[illegible] [illegible]cal-to-distributed transformation between the source and the target xtrees.

[illegible]he complete set of rules which constitutes the translation specification from t[illegible] [illegible]bset to S/R language includes 55 rules and takes 480 lines without the projectiv[illegible] [illegible]ar rules which are presented separately.

[illegible]he compiler itself is implemented, using the Karlsruhe Cocktail toolbox [7] whic[illegible] [illegible]neration of a tree-based structure in C language as an instance of the intern[illegible] [illegible]terface between the compiler front and back ends. In our case this interface is [illegible] [illegible]ommon Abstract Representation) for [illegible] 92. The compiler front end, which [illegible] SDL program and converts it to a *CAR* structure consists of a LALR-pars[illegible] [illegible]semantical analyser. The compiler back end takes 830 lines in Karlsruhe C[illegible]

pattern-matching facilities. This allows the straightforward i[illegible]ementation of su[illegible] (part of a) translation rule that only exploites the recursive descent driven by the stru[illegible] tural hierarchy of SDL constructs. In case of more intricate (parts of) translation rul[illegible] involving distributed-to-local (or local-to-distributed) transformations, we used multi-pa[illegible] code generation

References

[1] "Functional Specification and Description Language (SDL)", CCITT Blue Book, Recommenda[illegible] Z.100, Geneva, 1992.

[2] D. Bjorner, C.B. Jones (eds.), "Formal Specification and Software Development", Prentice-H[illegible] International. 1982.

[3] E. Bounimova, V. Kaufman, V. Levin, "Towards Language Mapping Description", Ve[illegible] Moskovskogo Universiteta, series Vychislitelnaya matematika i kibernetika, 4, 1978, 68-73 (in R[illegible]sian).

[4] E. Bounimova, "A Method of Language Mappings Describing", Dr. Dis., Moscow University, 19[illegible] (in Russian)

[5] E. Bounimova, V. Levin, "Specification rules for representation of an SDL subset in COS[illegible] Loose asynchronous model," Technical Report of the TÜBİTAK SR&DC, January 1996.

[6] P. Deransart, M. Jourdan, B. Lorho, "Attribute Grammars: Definitions, Systems and Bibliogr[illegible] LNCS no. 323, Springer Verlag, 1988.

[7] J. Grosch, H. Emmelmann, "A Toolbox for Compiler Construction", in D. Hammer (ed.), Co[illegible] Compilers'90 Third Int. Workshop. Springer Verlag LNCS 477 , 1990, 106-116.

[8] J. Hopcroft, J. Ullman, "Introduction to Automata Theory, Languages, and Comput[illegible] Addison-Wesley, 1979.

[9] G. Kahn, "Natural Semantics", In Proc. of the 4th Annual Symposium on Theoretical A[illegible] Computer Science. LNCS 247, 1987, 22-39.

[10] R. Kurshan, "Computer-Aided Verification of Coordinating Processes", Princeton Universi[illegible] 1994.

[11] V. Levin, "Formal Description of Programming Language Context Conditions", Dr. Dis., [illegible] University, 1978 (in Russian).

[12] V. Levin, Specifications Using the Data Type Syntactic Nodes, In: Hans Langmaa[illegible] Neuhold, Manfred Paul, eds., Software Construction - Foundation and Application, D[illegible] Seminar-Report, 29, January 1992, 22-22.

[13] V. Levin, E. Bounimova, O. Başbuğoğlu, K. İnan, "An Interface Implementation using [illegible] Compiler Specification Method ", Research Report of the TÜBİTAK SR&DC, November [illegible]

[14] M. Odersky, Defining Context-Dependent Syntax Without Using Contexts. ASM Trans. o[illegible] Systems, v.15, no.3, 1993, 535-562.

[15] A. Poetzsch-Heffter, Programming Language Specification and Prototyping Using the MA[illegible] PLILP'93 Proc., LNCS, 1993.

[8] and, additionaly, a library of PUMA's procedures consisting of 220 lines, which is specifi[illegible] [illegible] involved into our project

SSST (State-Space Search Tool): A Probabilistic Forward-Chained Expert System Shell with Full Backtracking

Zeki O. Bayram

Computer Engineering Department
Boğaziçi University
Bebek 80815/İstanbul-Turkey
internet: bayram@boun.edu.tr

Abstract

[illegible] describe a probabilistic backtracking forward-chained expert system shell th[illegible] [illegible]forms a best-first search of the state-space consisting of working memory states. Th[illegible] [illegible] space that needs to be traversed can be narrowed significantly through tactful use [illegible] [illegible] **context mechanism**. **Fail conditions** detect forbidden working memory states a[illegible] [illegible]se immediate backtracking. Heuristic information about which rules should ha[illegible] [illegible]her priority are encoded in the rules at the granularity level of **condition elements** [illegible] form of **importance factors**. Facts in the working memory have associated w[illegible] [illegible] **confidence factors,** which allows the representation of uncertain information. T[illegible] [illegible]adigm that results as the combination of this specific set of features permits declarat[illegible] [illegible]cification of the state space required for the solution of many kinds of scheduli[illegible] [illegible]blems and other kinds of problems requiring intelligent search of the state space w[illegible] [illegible]sible backtracking.

[illegible]ywords: Expert system shell, forward chaining, backtracking, context, heuris[illegible] [illegible]elligent search, inference

Introduction

[illegible]ward chaining inference engines start with an **initial state** of the world (as describ[illegible] [illegible] the initial contents of the *working memory*), and seek to reach a **goal state** throu[illegible] [illegible]eated application of a certain set of transformations, usually specified in the form of [illegible] rules [5,7]. The deficiency of most current forward chaining expert system shells [illegible] they allow a **single line of reasoning** (also called **hill climbing**). If, upon reaching [illegible]ain state the system cannot proceed any further (possibly because a wrong choice w[illegible] [illegible]de earlier in the selection of which rule to fire), the system stops without finding [illegible]tion. The main reason for this deficiency is efficiency: saving the *choice points* [illegible] iteration certainly consumes a lot of space.

[illegible] **State Space Search Tool (SSST)** is an expert system shell with an inference engi[illegible] [illegible] does allow backtracking. In SSST the efficiency problem is dealt with in three wa[illegible] [illegible] by making the search space small through a **context mechanism**, (2) by caref[illegible] [illegible]ing of branches in the search tree that cannot possibly lead to a solution using f[illegible]

EK 9

A Verifiable Software/Hardware Co-design Using SDL and Cospan

V. Levin
TÜBİTAK-BİLTEN
Ankara
levin@srdc.metu.edu.tr

E. Bounimova
TÜBİTAK-BİLTEN
Ankara
ella@srdc.metu.edu.tr

O. Başbuğoğlu
ASELSAN
Ankara
basbugog@srdc.metu.edu.tr

K. İnan
E&E Eng. D., METU
Ankara
inan@srdc.metu.edu.tr

Abstract

A new bilingual specification environment consisting of the two languages SDL and S/R is described. The new design is based on the SDL-to-S/R compiler that was implemented at a previous phase and introduces the necessary interface semantics as restrictions on special SDL shell processes that encapsulate the interfaced S/R modules. The environment can be used in SW/HW co-design by involving the verification engine COSPAN geared to the S/R language specification.

1 Introduction

The paper extends the scope of our bilingual approach initially introduced in [5] to describe SW/HW systems where the HW part is modeled as synchronous sequential logic composed of standard blocks (gates, registers) interconnected to each other. In this paper we generalize this approach to describe and illustrate a specification formalism for distributed software (SW) operating under possibly synchronous environment which may involve HW or lower level software or any other environment that is only suitable for specification by the other language — in our case the synchronous language S/R — of the bilingual formalism. The distributed software is specified in SDL [1] and the environment consisting of synchronously operating components that are awkward or impractical to be specified by SDL, such as hardware (HW), are specified by the complementary language S/R of the bilingual formalism[1].

In order to illustrate an abstract concept of a synchronous environment we chose a simple abstract machine operating under a real time constraint. This example also illustrates the hard time constrained system verification via the timing module of COSPAN as applied to an SDL specification although, in general, SDL is known for its lack of well-defined time semantics. Below we point out three distinct reasons for choosing SDL and S/R as the language components of our bilingual environment.

First, whereas SDL is an asynchronous language that is specifically suitable for communicating software component modeling, S/R is a synchronous language that, for example, can easily model sequential circuits based on state equations (like in VHDL) that operate synchronously under the control of a single clock. This makes the bilingual environment particularly attractive for SW/HW co-simulation or co-verification.

Second, S/R has the features to simulate asynchronous and even nondeterministic behaviour and can therefore model higher level abstractions if necessary. Indeed it is precisely for this reason we decided to implement a compiler from SDL to S/R to start with [4]. This feature is a definite advantage of the bilingual specification environment since the specified code is compiled into pure S/R and then can be verified — as opposed to mere simulated — using the powerful COSPAN tool. In this sense VHDL does not seem to be a substitute for S/R since the former has neither the higher level features like S/R nor a verification engine like COSPAN. This is the basic reason behind some of the shortcomings of the recent work in SW/HW co-design [3, 7, 8], namely that the tools are restricted to simulation and cannot perform exhaustive verification.

Finally, our contacts with the industrial users of SDL have revealed that SDL specifications frequently contain an environment part which cannot be specified by SDL — usually lower level interface parts — and therefore the corresponding simulation platforms either require manual intervention or are simplistically modeled which may restrict the scope of the validation exercise. Our bilingual formalism that offers S/R as an additional specification language together with an interfacing notation is a remedy for this problem since both simulation and verification can then be carried out without the previous restric-

[1] S/R (select/resolute) is an input language for the verification tool COSPAN. Both the language and the tool are products of Bell Labs [6].

COST 247 International Workshop on Applied Formal Methods in System Design, Maribor, June 17–19, 1996

tions.

In general the advantage of having a second language like S/R which has both low and high level linguistic characteristics is a general asset in writing specifications for design. In any context — that cannot always be predicted — where using SDL is either impossible (such as modeling a finite state machine!) or at least difficult and awkward, S/R becomes a desirable, complementary substitute.

In the next section we briefly outline features of the languages SDL and S/R with a special emphasis on S/R since it is both less well-known and has not-so-familiar features that are relatively alien to a software engineering audience. In section 3 we describe the bilingual formalism and in sections 4 and 5 we illustrate our design via two simple examples that involve a modular hardware alternating bit (AB) protocol driven by software and a resource allocating system subject to hard timing constraints, respectively. In the first example the hardware is specified in S/R, the driving software is specified in SDL and the SW/HW interface is specified by a special SDL process called a *shell* process, the semantics of which differs from ordinary SDL processes. In the second example the clients are modeled by SDL software processes whereas the server is modeled as an S/R process.

2 The languages SDL and S/R

SDL (specification and description language) is an ITU-T/ISO standard language [1] used especially by telecommunication industries for design and implementation of software for embedded systems. There are both textual and graphical syntactic definitions of SDL and the latter visual facility has made it popular among the telecommunication software engineers as a relatively popular user-friendly tool for design and implementation. SDL is a prime example of the reactive programming paradigm whereby independently executing SDL processes progress one-at-a-time by reading their input message reception queues and *react* to these messages by making internal assignments and by sending new messages to other processes' input queues. The reader is welcome to view the graphical syntax given in Figure 3(c) as the flow chart of a typical program segment in some ordinary programming language.

Unlike SDL which is an asynchronous language with interleaving semantics, S/R is a synchronous formalism within which all processes move simultaneously under an imaginary global clock pulse referred to as *crank* in S/R. The S/R language enjoys the control theory paradigm in which component dynamical systems (S/R processes) are represented by state equations :

$$x(n+1) = f(x(n), u(n))$$
$$y(n) = g(x(n), u(n)) \quad (1)$$

Here n counts cranks — imaginary clock pulses, mentioned above. $x(n)$ is a vector denoting the values of the state variables just after the nth crank — henceforth referred to as the nth instance. $u(n)$ and $y(n)$ are the vector of input and the output values respectively at the nth crank. The inputs are of two kinds. Some are simply outputs of other processes which are used as external inputs to the process. In the S/R context, the latter process gets the access to output variables of other S/R processes via a declaration **import** *variable_name_list*. This is the mechanism of interprocess synchronization in S/R. In conventional (control theoretic) terms this is simply interconnection of outputs of dynamical system blocks to the inputs of other such blocks. The second kind of input is the internal input where the value is allowed to be nondeterministically selected.

In S/R each variable is declared either as a state variable (**stvar**) or a selection variable (**selvar**). The input $u(n)$ and output $y(n)$ variables are modeled by selection variables. In view of the state equation model above the assignment of each state variable corresponds to the next state function which is of the form

$$x_j(n+1) := f_j(x_1(n), \ldots, x_k(n), u_1(n), \ldots, u_m(n)) \quad (2)$$

where the general format and the associated S/R syntax for the function f_j is given as

$$\textbf{asgn } x_j \to value_1?guard_1|\ldots|value_N?guard_N \quad (3)$$

where all the values and guards (predicates) are possible functions of the state vector $x(n)$ and the input vector $u(n)$ and '|' denotes the choice function[2]. In a similar syntax, assignments are also made to selection variables implementing the output equations

$$y_j(n) := g_j(x_1(n), \ldots, x_k(n), u_1(n), \ldots, u_m(n)) \quad (4)$$

where the associated S/R syntax for this kind of assignment is given by

$$\textbf{asgn } y_j := value_1?guard_1|\ldots|value_L?guard_L \quad (5)$$

Whereas recursion is allowed and consequently there is a one step delay in state assignments, no recursion

[2]In S/R the guards exhaust all possibilities by an explicit or implicit *else* function so that no f_j is a partial function. On the other hand, if guards are not mutually exclusive f_j is in fact a relation and further nondeterminism prevails.

is allowed for the output assignments and the current output values are simply read-out as a memoryless function of the current state and input variables.

3 SDL&S/R interface formalism

In this section we decribe the *SDL&S/R interface formalism* for specifying systems in a bilingual environment. In this formalism every S/R process which has a direct interface to the software (or SDL) part is encapsulated within an SDL-like process called a *shell*. The remaining processes are specified either in proper SDL or proper S/R. In what follows we shall use a terminology that assumes that S/R processes correspond to hardware and SDL ones to software. In view of our remarks above the S/R part could also be interpreted by some suitable entity that can be viewed as an environment interacting with the SDL part.

The SDL&S/R notation used to specify a shell can syntactically be characterized as a restricted SDL with S/R *adaptation pragmas*[3]. The adaptation pragmas are utilized in the SDL context as comments beginning with keyword S/R. They are interpreted as semantically meaningful statements that modify the standard SDL semantics of the shell process to coordinate executions on the two sides of the interface: namely, the shell SDL constructs and the encapsulated S/R process constructs. The modifications involved in the shell semantics consist of restrictions to normal SDL semantics and in that sense is a refinement of SDL that applies only within a shell process.

The interactions of a shell process with other shell or non-shell SDL processes are governed by exactly the ordinary SDL semantics. The restrictions mentioned above apply only to the interactions with the interfacing hardware modeled by the encapsulated S/R process. The basic idea involved is the management of the special variables *shared* by the shell SDL (software) process and the encapsulated S/R (hardware) process.

The shell communicates with the encapsulated S/R process as if it itself is an S/R process. So, the two, viewed both as S/R processes, simply share some selection variables. More precisely, the shell imports from the encapsulated S/R process some of its selection variables which are referred to as *interface* variables. These variables are classified into *input* and *output* variables (relative to the encapsulated S/R process) with the following assignment restriction: input and output variables can only be assigned values within the shell process and the encapsulated S/R process, respectively. In the shell, which is viewed as an SDL process, the interface variables of both (input and output) types are declared as its local variables with defining occurrences followed by the adaptation pragma `/*S/R*/`. Input and output variables are distinguished from each other simply by the use of an SDL initializing expression which is made compulsory for the former case and forbidden for the latter case.

The interface variables must be provided with values. Therefore, we allow for the shell and the encapsulated S/R process to share data types. It is possible to define the type of a shared variable either in the shell or in the encapsulated S/R process. A shared variable data type can be declared in the shell with adaptation pragma `/*S/R*/`. Then the associated semantics implies that in the post-compilation phase the encapsulated S/R process is forced to use this type within its own declaration of this shared selection variable. Alternatively the definition of the shared data type can be declared in the encapsulated S/R process[4]. In this case we allow for such an already declared data type, say T, to be re-declared in the shell without contents : `newtype T endnewtype; /*S/R*/`. In the compiled S/R context, this data type is filled with the contents extracted from its S/R definition given in the encapsulated S/R process.

The shared access to an interface variable means that whenever this variable is assigned a value within the shell or the encapsulated S/R process its value is immediately reflected in the other process. We therefore extend S/R's selection/resolution crank (imaginary clock cycle) semantics partially to the shell process to allow it to coordinate with the encapsulated S/R process. This is accomplished as follows.

In the shell we impose on a task (which is a box with assignments in the graphical format of SDL) an *atomicity constraint* to tightly bind its execution to the S/R crank cycle. So, in the compiled S/R image of the shell process all the assignments contained within one task execute at the same crank and different tasks execute at different cranks. We do not allow for a task box to contain repeated assignments to the same variable. Any recursion in the assignment of values to the interface (hence, input) variables is also forbidden. These two restrictions on the assignments contained within one task box simply follow the S/R assignment rules for selection variables explained in

[3]The word *pragma* was first used in Algol 68 to mean a semantically half-alive entity: it does not have an independent semantics but it influences compilation by modifying the semantics of another phrase which it qualifies.

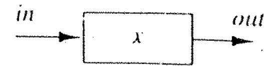
[4]When the hardware environment is modular with standard modules this would be the usual case.

the previous section.

Another mechanism to coordinate the shell behaviour with the encapsulated S/R process is devised via *interrupt signals*. From the S/R viewpoint, a hardware interrupt is implemented by a predicate that guards a transition handling the interrupt. In the shell process we denote, in SDL notation, a synchronous interrupt by a priority input box with an enabling condition where the latter stands for the interrupt predicate used in S/R as mentioned above. The interrupt signal expected at this priority input box must be specified in the **signalset** list of the shell and marked by the adaptation pragma S/R. Instances of this special interrupt signal must not appear anywhere else in the SDL environment since it is not a genuine SDL signal and only a construct used for this interrupt mechanism. The new semantics of the interrupt signal input is such that the immediately following assignment transition starts executing whenever the enabling condition appended to the input box evaluates to *true*. Since this enabling condition can contain output variables which are assigned values within the encapsulated S/R (hardware) process, this semantics models hardware interrupts in the same manner as in the S/R language. In order to allow the interrupt input to coincide with other standard SDL inputs coming from the software environment we forbid other priority inputs — but allow normal ones — at a state where interrupt inputs are used.

It may be important to assign values to hardware input variables at the same S/R crank at which an interrupt appears. So we assume that the first action in the immediately following transition starts executing at the same crank as the interrupt schema descibed above issues (and consumes) an interrupt signal. This requirement together with the atomicity imposed on a task allows for modelling conventional interrupt handling.

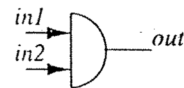
Finally, we generalize the coordination semantics of the shell. First, in the shell, including its procedures, an action is restricted to be a task, a decision, an output, a procedure call or a timer setting or resetting. Second, an expression can contain a procedure call only if the corresponding procedure body contains neither states nor actions, but only returns an expression. Any expression in any context is assumed to execute in one crank cycle. The first action in a transition triggered by evaluating a decision starts its own execution in the same crank within which the decision was evaluated.



$x(n+1) = in(n)$
$out(n) = x(n)$

```
proc  D
  import in  /*BinarySignal*/
  stvar  x: BinarySignal
  selvar out: BinarySignal
  init   x:=0
  asgn   x ->in
  asgn   out:=x
end  D
```

Figure 1: D(flip-flop)



$out = in1*in2$

```
proc  AND
 import  in1, in2
 selvar  out: BinarySignal
 asgn  out:= 1 ? (in1=1)*(in2=1) |
                0
end  AND
```

Figure 2: AND

4 Example for SW/HW co-specification: AB protocol

In this section we illustrate the bilingual specification formalism via an example in which software specified in SDL makes use of the hardware AB protocol to transmit a number of — in the example three! — messages from the sender to the receiver side. The AB protocol is specified in the S/R language and presented by the interconnection of standard hardware circuit diagrams that implement the sender and receiver modules given in Figures 10(a) and 10(b) — see Appendix — as well as the imperfect intermediate channel and the timer whose details are omitted in the paper.

In order to explain the operation of the hardware AB protocol we first direct the attention of the reader to the state equation representation of some standard hardware blocks used in the example and we present the corresponding S/R process code in the manner explained in section 2. We start with the memoryless gates such as AND, OR, XOR, INV where INV means logical inversion. The state equations for the gates have no state variables which would model memory.

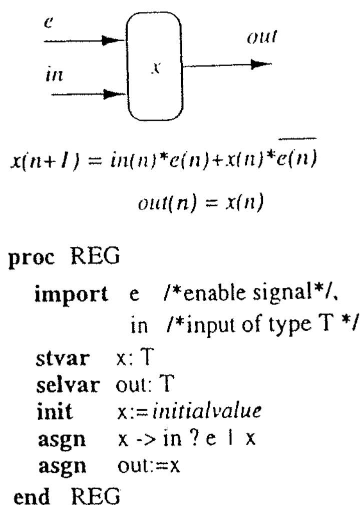


Figure 3: REG

So, the gate AND is defined by only the output equation of the type 4 (see section 2) which is explicitly expressed as $out(j) = in1(j) * in2(j)$ where $*$ stands for the logical AND operation. The corresponding S/R code is given in Figure 2. The state equations and S/R code for the other memoryless gates are straightforward. On the other hand the simple delay circuit D is defined by the state equations with a single state variable (memory) which are given together with S/R code in Figure 1. Note that the state and output assignment semantics are distinguished in this case by delayed and nondelayed left-hand sides. Finally the register buffer REG is defined by the state equations and S/R code given in Figure 3. Note that unless the enabling bit e is high the value of the register contents remains unaltered.

The operation of the hardware sender can now be described in terms of the sequential logic of the circuit diagram given in Figure 10 (a) in Appendix. Whenever the enabling bit e is high, register REG accepts the value of the *Msg* input. Then this value becomes available together with the <*AB_tag*> output for transmission to the channel when the <*Msg_available*> bit is high. The three inputs to the gate OR1 represent three separate occasions which raise the <*Msg_available*> bit and hence initiate a transmission. These occasions are: a new message is available or a timeout message is received or an acknowledgement is received with the incorrect *ABit* (stands for 'alternating bit'). The gate XOR1 mechanizes the bit alternation facility. Whenever the received *ABit* differs from the previously sent *ABit* given by *Transmitted_ABit* — which means successful transmission is completed — then the output of

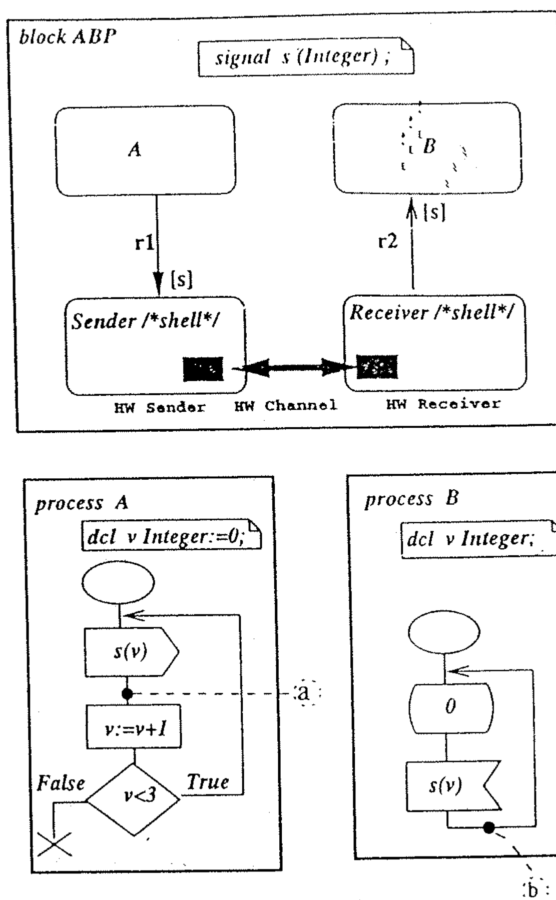


Figure 4: ABP: SW part

XNOR1 and therefore with a single step of delay the output of D1 (<*Ready*>) are both high, thus initiating the next message to be sent by the user. The internal loop between gates XOR1 and XNOR1 operates in the manner desired: if output of XNOR1 is high then the next value of *Transmitted_ABit* is replaced by its complement. The *Chan_ack* signal which is the next *AB_tag* expected by the receiver is received from the channel whenever the signal *Chan_ack_avail* is high. Note that the latter signal can be also used to model imperfect (lossy) channel by invoking the internal nondeterminism within the S/R model of the channel as explained in section 2. A similar explanation also prevails on the receiver side which the reader can easily follow from Figure 10 (b) given in Appendix.

Now we explain the user software that activates the hardware AB protocol. This software is specified in SDL in two parts: the part that does not interact directly with the hardware and the part that describes the SW/HW interface, namely the SDL shell processes. The overall architecture is given in Fig-

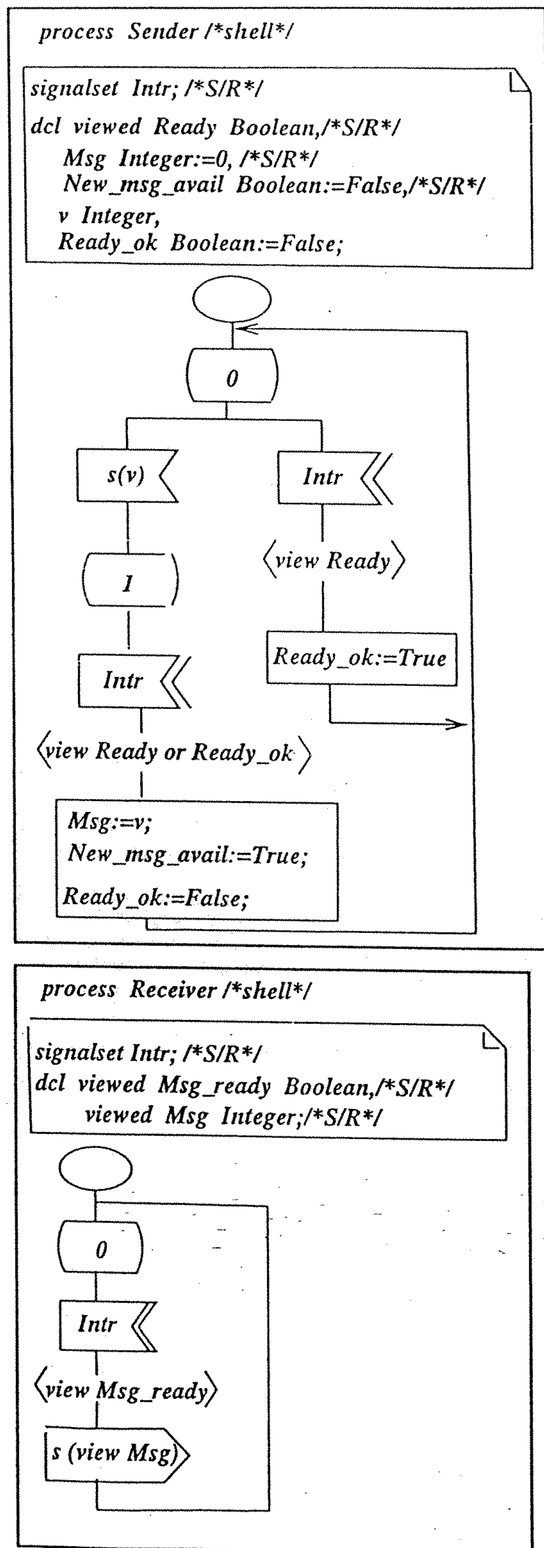
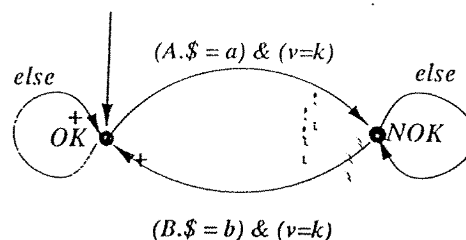


Figure 5: ABP: shells



Liveness property:

Each message sent by A should eventually be received by B

Figure 6: Verification task

ure 4. The hardware part described in S/R for the sender, receiver and the channel are shown by the darkened boxes and arrow and the four SDL processes play the role of user intentions specified by A and B (user sender and user receiver) and the respective shells that encapsulate the sender and receiver hardware. The entire function of the shell processes, as explained in the previous section, is to define the interface between the software and the hardware and to coordinate the demands coming from and responses going to other SDL processes.

The user SDL processes specified as A and B are given in Figure 4. The process A simply requests to send three consecutive messages with values 0,1 and 2 to the receiver site and then halts. On the other hand the receiving process B simply receives what is coming from the outer world.

The respective shell SDL processes *Sender* and *Receiver* are given in Figure 5. The *Sender* can read its input buffer for signal s carrying the current value of v. *Sender* goes to the state 1 where it waits for the sender hardware interrupt signal *Intr* specified by the enabling condition *Ready*. When the interrupt occurs, *Sender* transfers the received value v to the sender hardware input *Msg* together with the enable high hardware input e, using the special task with assignments as explained in the previous section. The second branch at the state 0 is required because hardware interrupt signal can arrive earlier than software input. The *Receiver* process can easily be understood in a similar way from Figure 5.

In our exercise, the hardware AB protocol is specified in S/R and comprises of 33 S/R processes modelling hardware blocks of standard types shown in Figures 1-3. The software and shell processes given in Figures 4,5 are specified in SDL, compiled into S/R and then their S/R code is linked to the AB protocol

S/R code. So, the entire SW/HW system is converted into some COSPAN ω–automaton [6], thus allowing verification using the COSPAN verifier. We have applied a specific verification technique [2] linked to the original SDL source of the COSPAN ω–automaton.

In order to verify the system, we have specified some COSPAN *verification tasks* which express system requirements. In Figure 6, an example of the verification task is presented for a liveness property: each message v delivered from the user process A should eventually be received by the user process B. This verification task is formalized by a two-state acceptance automaton synchronized to the system automaton via coupling of boolean predicates [6]. These are predicates RA and RB in Figure 6 that specify the preconditions for a transition from one state to another. RA means that process A delivers message v — i.e., A reaches the location a in the process transition structure (see Figure 4, process A); RB means that process B receives this message — i.e., B reaches the location b in its transition structure (see Figure 4, process B).

The COSPAN verifier detects a violation of a system requirement if the ω–automaton of the system contains an unspecified cycle. So, a verification task must specify all the harmless cycles in the system. One mechanism that facilitates cycle differentiation is by using *recur edges*, marked by the sign '+' . In a system cycle that contains a recur edge the verification does not fail. For example, in Figure 6 only the NOK self-cycle is left as harmful (*bad*) in the sense that it is not protected by any recur edge. In view of the fact that all other cycles are protected by recur edges, the bad cycle will be detected by COSPAN in case a message is sent from A, but never received by B. For our system, the task in Figure 6 performs for each v=0,1,2 without any violations.

5 Example of a bilingual specification with timing constraints

In this section we illustrate by a simple example how the bilingual SDL&S/R formalism can be used for specifying a combined system with timing constraints. Since it is the Cospan verifier that does the timing verification, we take care to express the timing constraints in SDL in a manner that is suitable for the Cospan verifier.

The example system shown in Figures 7, 8 comprises of a HW server providing some service, two identical SW clients of this server and a shell via which the clients get an access to the server.

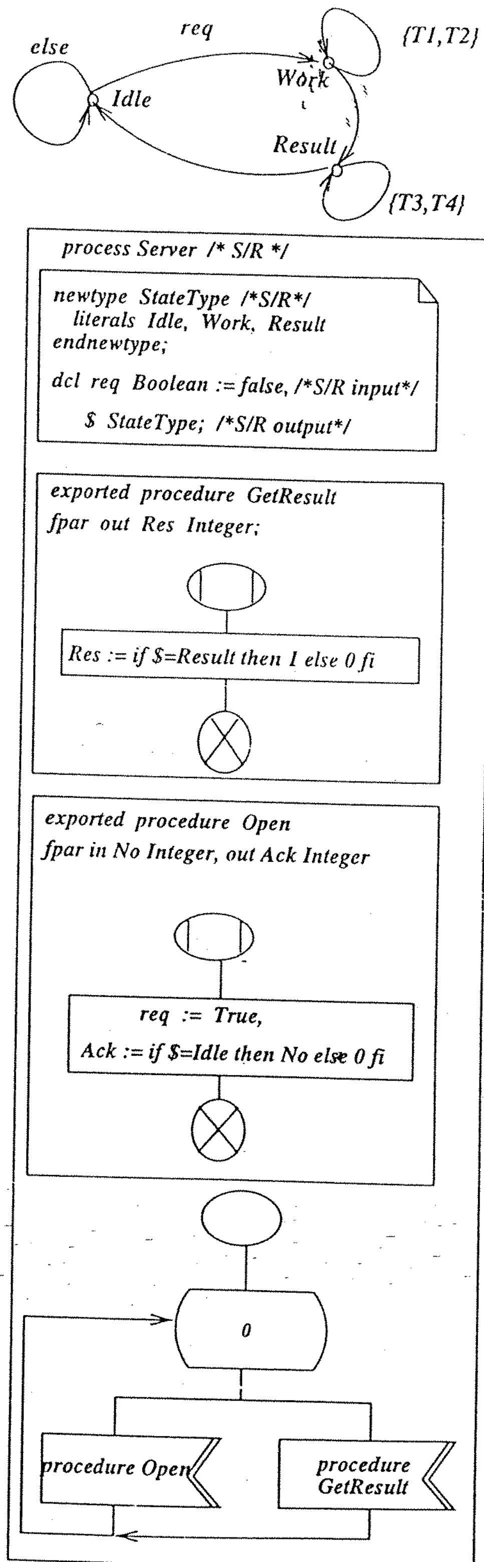


Figure 7: A timed HW model and its shell

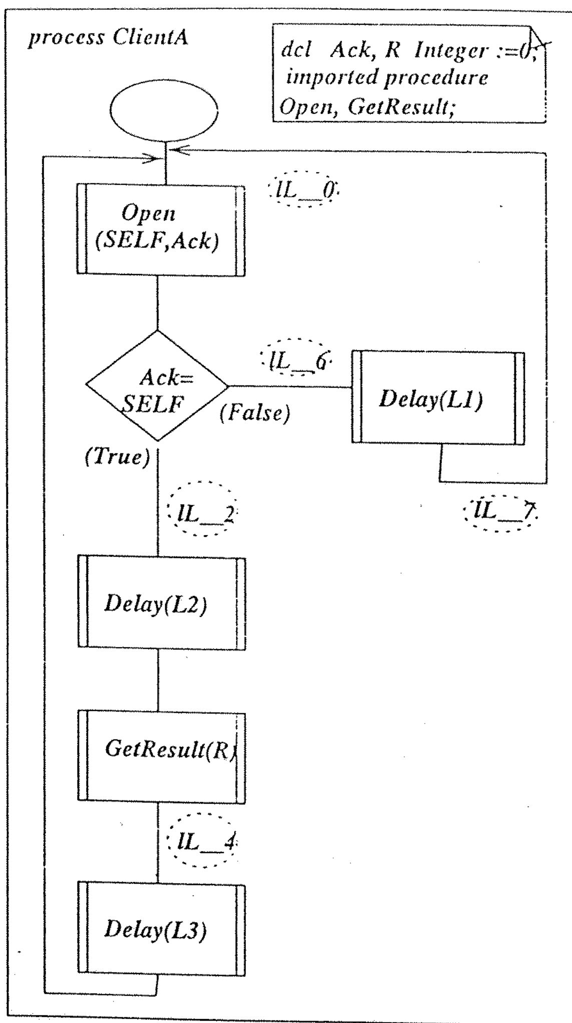
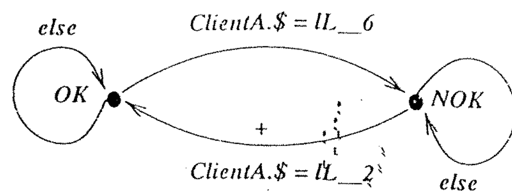
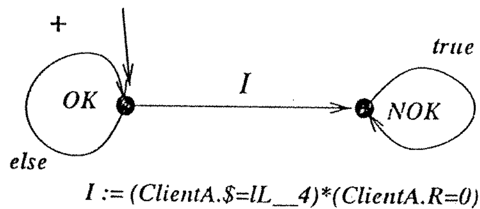


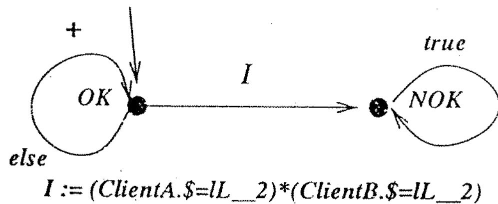
Figure 8: SW Client



(a) Liveness property:
each Client eventually gets access to the Server



(b) Safety property:
each Client gets its Result from the Server



(c) Safety property:
each Client cannot be served simultaneously

Figure 9: The verification tasks for the timed model

The server is modeled in S/R as a three-state automaton. It waits for a request from a client at the state *Idle*. After receiving a request, the server moves to the state *Work*, where it supplies the requested service within a time duration of $t1$ and then moves to the state *Result*. The client can then read the produced result within a time duration $t2$ after which the server comes back into the state *Idle*. The client's request which triggers the transition of the server from *Idle* to *Work* is the *True* value of the server selection variable *req*. The symbol *req* stands for the input variable of the server shared with the shell process — which also is called *Server*. During each three-phase service cycle durations $t1$ and $t2$ are nondeterministically selected within the intervals $\{T1, T2\}$ and $\{T3, T4\}$, respectively, which are specified at the states *Work* and *Result*.

The server shell is specified in the SDL&S/R interface formalism described in section 3. However, for this example we used more developed interface features, namely, SDL synchronous communication facility provided by *remote procedures*. We remind that a remote procedure is a procedure declared in its owner process as *exported*. This means that it can be remotely called from other process(es). Although the remote procedure call is formally implemented via the basic SDL asynchronous signal exchange — using implicit SDL channels — it obeys a protocol that guarantees the following synchronization: the calling process does not move from the point where it calls the procedure until the procedure exporter process receives the implicit signal with the procedure call, serves the call, and sends the result back to the calling process by an implicit signal.

The SW part of the example system consists of the two identical client processes one of which is specified in Figure 8 as the *ClientA* process. *ClientA* progresses by first sending a request to the server

shell (by calling the remote procedure *Open*), and then checks the server's acknowledgement, either repeats the request to the server or, first gets the result from the server and second, repeats the request to the server. Associated with these actions are time delays symbolized by procedure boxes named *Delay* as shown in the SDL diagram of Figure 8. Such a delay procedure can be implemented in SDL with a delay parameter t by setting some timer to the value of $NOW+t$, where NOW is the current time and waiting for the *Timeout* signal at a state where all the other signals are saved. We assume that all the actions involved into this procedure (timer setting, signal saving, timeout signal consuming) have 0 duration.

The way that the timing constraints induced by the delay procedures translate into Cospan is accomplished by first noting the actual Cospan states corresponding to the SDL pseudo states shown by dashed circles (e.g. *IL__0* to *IL__7* in Figure 4) and then introducing lower and upper bound for the time delays at those Cospan states that correspond to SDL pseudo states before the delay procedures.

The shell process is presented in Figure 7 in graphical SDL. It includes declaration of shared variables *req* (input) and *$* (output). The variable *$* gives the shell an access to the current state of the HW server (*Idle*, *Work* or *Result*). Declaration part of the shell process also includes remote procedures *Open* and *GetResult*. In the transition part of the shell process, there is only one state *0*, in which it serves procedure calls.

Procedure *Open* sets the shared variable *req* of the HW server to *True* (request is received), and sets an acknowledgement *Ack* into the number of the requesting client, provided the server is in its *Idle* state, which is signified by the value of *$*.

Procedure *GetResult* sets its output parameter *Res* to 1, provided the server is in its *Result* state.

The final S/R code is verified using the Cospan verifier and the tasks given in Figure 9 corresponding to desired safety and liveness properties. Whereas there is only one liveness property to be verified that corresponds to nonstarvation and given as a task in Figure 9(a), there are two safety properties given by Figures 9(b) and 9(c) corresponding to correct response by the server and mutual exclusion, respectively.

Since this is a verification task with hard time constraints, the actual values of the timing parameters have to be supplied to Cospan. The underlying verification method for timed systems and the models used for timed automata in Cospan is beyond the scope of this paper and the reader is referred to [6] for a detailed treatment.

For the values

$$T1 = 1,\ T3 = 2,\ L1 = 4,\ L2 = 2,\ L3 = 3$$

all the safety and liveness properties were succesfully verified. On the other hand, if all the timing parameters were set to 1, verification of the liveness property and when they were set as

$$T1 = 2,\ T3 = 2,\ L1 = 1,\ L2 = 1,\ L3 = 1,$$

verification of the safety property given by 9(b) failed. Since the example is simple enough, the reader can independently justify these results by simple calculation.

6 Conclusion

We have presented a bilingual SDL&S/R interface formalism. It allows specifying a combined software/hardware system in a natural way: software components are specified in SDL, hardware components are specified in S/R and the design of the SDL-based notation to specify the interfaces between software and hardware components semantically extends SDL though being a syntactic restriction of SDL. The semantic extension of SDL allows for a special shell process to share variables coupling architecture between the encapsulating shell process and the encapsulated hardware S/R process. The shared variables are precisely those representing the interface registers for writing and reading data to and from the hardware.

From the implementation point of view, the bilingual SDL&S/R formalism has been designed in a way that can utilize the existing compiler from SDL into S/R [4]. In other words, once the bilingual specification stage is complete, the code can be converted into a pure S/R code making use of the existing compiler from SDL into S/R with the necessary additions to incorporate the interface semantics explained in the paper. The ability to convert the bilingual specification code into a monolingual S/R code is a definite advantage of the design presented since this allows the verification of the co-design using the powerful Cospan verification tool which is currently undergoing a transformation to become a full commercial product of Bell Labs.

References

[1] "Functional Specification and Description Language (SDL)", *CCITT Blue Book, Recommendation Z.100*, Geneva, 1992.

[2] E. Bounimova, V. Levin, O. Başbuğoğlu, K. İnan, "A Verification engine for SDL specification of communication protocols", in: *S. Bilgen, M. U. Çağlayan, C. Ersoy (eds), Proceedings of the First Symposium on Computer Networks*, Istanbul, May 30-31, 1996, pp. 16-25.

[3] A. Boujarwah, I. Ahmad, et al, "Hardware/software codesign in the ESTELLE and VHDL environments", *Computer Standards & Interfaces*, Vol.17, pp.253-276, 1995.

[4] V. Levin, E. Bounimova, K. İnan, "An Interface Implementation using a Formal Compiler Specification Method", to be published in ISCIS_XI, 1996.

[5] V. Levin, O. Başbuğoğlu, E. Bounimova, K. İnan, "A Bilingual Specification Environment for Software/Hardware Co-design", to be published in ISCIS_XI, 1996.

[6] R. Kurshan, *"Computer-Aided Verification of Coordinating Processes"*, Princeton University Press, 1994.

[7] J. Witrebowisz, "Hardware Specification Generated from ESTELLE", in: *P. Dembinski, M. Sredniawa, eds.: Protocol Specification, Testing and Verification (PSTV'95)*, 1995, pp. 417-432.

[8] L. Pirmez, A. Pedroza, et al, "A methodology for the implementation of protocols in hardware from a formal description", in PSTV'95, 1995, pp. 401-416.

2nd International Workshop on Applied Formal Methods in System Design
Ignac Lovrek (Ed.)
June 18-19, 1997, Zagreb, Croatia

Ek 10

Verification using Leap Automata

H. Yenigün
Bell Laboratories
Lucent Technologies
700 Mountain Avenue, Murray Hill
NJ 07974, USA
husnu@research.bell-labs.com

K. İnan
Electrical & Electronics Eng. Dept.
Middle East Technical University
Balgat-Ankara
Turkey
inan@srdc.metu.edu.tr

Abstract

In [11], a transition leaping mechanism of [6, 5, 10] is presented as an additional and powerful complexity reducing mechanism used together with partial order methods for finite state verification. In this paper we improve on the original leaping approach by setting it in a more general context thereby enhancing the potential for greater complexity reduction. In particular by allowing ample sets that are not necessarily disjoint we present leaping results both for safety and liveness verification where the latter is presented only under a fairness assumption.

1 Introduction

Partial order methods exploit the interleaving redundancy for finite state systems and use a smaller fraction of the reachable states for the purpose of verification such as deadlock detection, model checking etc. [7, 9, 1]. Notwithstanding different versions of the partial order approach each with its own terminology they all share the trace equivalence definition that is based on independence of transitions - that generates the interleaving redundancy - and properties that are robust relative to the equivalence classes.

Recently a simple, yet powerful mechanism has been suggested as an approach that further reduces the effective search space for verification [11]. The approach in [11] is based on defining synchronous transition groups called *simultaneously executable sets* [5, 6] and *leap sets* [10, 11] that consist of a set of *independent* transitions such that not only the order in which these transitions are executed but also the intermediate states reached by individual executions bear no relevance to the final outcome of the verification exercise. Hence one can get away with executing these transitions in any, but only one such, fixed order - saving in total execution time - and ignoring the storage of the intermediate states of the executions - saving in both memory and time - without losing any information relevant to the verification verdict.

In this paper we extend these results to a more general setting. In particular we define maximal leap sets that allow for ample sets (in our choice of terminology *persistent sets*) that are not necessarily disjoint. We first define a unique leap automaton of a given automaton corresponding to a given *complete* collection - i.e., a set of persistent sets such that every enabled transition is covered at least by one persistent set - of persistent sets at each state of the original automaton. We exhibit the nature of leaping based on the richness of the set of persistent sets at each state, that underlies the leap automaton. In particular we show that, in general, the richer the persistent set collections, the greater is the leaping, hence complexity reduction, potential. Our subsequent results essentially state that the original automaton can be replaced by its simpler leap automaton where the properties to be verified remain intact.

In the next section we present the basic concepts and definitions culminating in the definition of a leap automaton. In section 3 we present our main results. First we show how to generate the appropriate initial maximal leap set at any state for a given finite or infinite string originating at that state. By an iterative use of this result we generate the sequence of leap sets for the leap automaton for a given finite string that terminates in a deadlock (Theorem 1 on safety verification) or an infinite fair string (Theorem 2 on liveness verification) of the original automaton. We briefly discuss our results in the Conclusions section.

2 Preliminaries

Let $A = (Q, \Sigma, \xrightarrow{a}, q_0)$ denote a deterministic automaton where Q is the state set, Σ is the event alphabet set where the elements of Σ label the transitions, $\xrightarrow{a}: Q \rightarrow Q$ is a partial (transition) function

for each $\sigma \in \Sigma$ and $q_0 \in Q$ is the initial state. We use the word 'transition' to mean the label σ of a transition. Hence, for example, we say that the transition σ is *enabled* at a state q if for some q', $q \xrightarrow{\sigma} q'$. For this we use the shorter notation $q \xrightarrow{\sigma}$. We use the same notation for the transitive closure, hence we write $q \xrightarrow{s} q'$ where $s \in \Sigma^*$ as a short-cut notation for consecutive transitions of the string s.

Next we define a dependency relation between the transitions of an automaton A and a string equivalence based on this definition ([4]).

Definition 1
A symmetric and reflexive relation $D \subseteq \Sigma \times \Sigma$ is said to be a dependency relation *iff for two transitions $\sigma, \sigma' \in \Sigma$, $(\sigma, \sigma') \notin D$ implies that for every $q \in Q$:*
(1) $q \xrightarrow{\sigma'}$ if and only if $q' \xrightarrow{\sigma'}$ whenever $q \xrightarrow{\sigma} q'$ and vice-versa when σ and σ' are interchanged,
(2) If $q \xrightarrow{\sigma}$ and $q \xrightarrow{\sigma'}$ then $q \xrightarrow{\sigma.\sigma'} q''$ iff $q \xrightarrow{\sigma'.\sigma} q''$.

Definition 2
Two transitions σ and σ' are said to be independent *if $(\sigma, \sigma') \notin D$. Two strings s and s' are said to be* equivalent, *denoted by $s \equiv s'$, if one of the strings can be obtained from the other by a sequence of permutations of adjacent and independent events in it.*

Remark 1
1 - Note that by inductive application of condition (2) of Definition 1, it can easily be seen that if s and s' are equivalent and $q \xrightarrow{s} q'$ and $q \xrightarrow{s'} q''$ then $q' = q''$.
2 - In applications A is a product machine (of communicating sequential processes) and the structure of each transition is in terms of synchronizations of the processes that participate in it and interleaving of processes that do not. Intuitively we can assume that if q_i is the ith local component of a global state and Γ_i is such that $q_i \xrightarrow{\sigma}$ if and only if $\sigma \in \Gamma_i$ then no two pairs of transitions in Γ_i can be independent. In a number of models (e.g. SDL [8]) either only one transition at q_i is enabled or any of the enabled transitions, after execution, pre-empts the others - such as in reading from the input queues in SDL - and hence local transitions are always dependent among themselves. Independence and choice occurs for the transitions of the product machine when they belong to the enabled transitions of different component processes.

Next we define the *persistency*[1] of a set of transitions, that is, a set of transitions that remain enabled along all paths that start from some current state and move via transitions outside this set.

[1]The term persistent set is due to [1] and is based on the notion of faithful decomposition given in [2].

Definition 3
Let q be a given state of A, Δ be the set of all enabled transitions at q and $\Gamma \subseteq \Delta$. We say that a non-empty Γ is a persistent *set iff for every string s of the form $s = \sigma_1 \ldots \sigma_k$ with $\sigma_i \notin \Gamma$ for $i = 1, \ldots, k$ and $q \xrightarrow{s}$ it follows that σ_k and γ are independent for all $\gamma \in \Gamma$. A set of persistent sets $\mathcal{G}$ at the state q is said to be* complete *if $\Delta = \cup_{X \in \mathcal{G}} X$.*

Any set of persistent sets that include the entire set of enabled events (which clearly is a persistent set) is a complete set. Another conceptually simple - but computationally burdensome ! - example of a complete set is the set of all persistent sets.

Definition 4
Let q be a given state of A, Δ be the set of all enabled transitions at q, $\mathcal{G}$ be a complete set of persistent sets at q. $\Theta \subseteq \Delta$ is called a leap set *(of transitions) at q relative to $\mathcal{G}$ if for each $\theta \in \Theta$ there exists a persistent set $\Gamma_\theta \in \mathcal{G}$ such that $\theta \in \Gamma_\theta$ and $(\Theta \setminus \{\theta\}) \cap \Gamma_\theta = \emptyset$. The leap set Θ is called* maximal *if it has no strict superset that is a leap set.*

We can now define the leap automaton $Leap(A)$.

Definition 5
The leap automaton relative to a given collection $\{\mathcal{G}(q)\}_{q \in Q}$ of complete persistent sets defined at each state q corresponding to a given automaton $A = (Q, \Sigma, \xrightarrow{\sigma}, q_0)$ is defined by $Leap(A) = (Q, 2^\Sigma, \xrightarrow{\Theta}_L, q_0)$ where $q \xrightarrow{\Theta}_L q'$ iff $q \xrightarrow{s} q'$ where s is a string composed of concatenations of elements of the set Θ in any *order where Θ is a maximal leap set of transitions at q relative to $\mathcal{G}(q)$.*

Remark 2
If Θ is a leap set at some state q then all the transitions in Θ are pairwise independent, hence any two strings obtained by concatenating the events in Θ using different orderings are equivalent.

3 Main Result

In this section we present the main result in terms of two theorems that relate an automaton A to its corresponding leap automaton $Leap(A)$. The latter is a formalism which acts as a simplifying complexity relief mechanism since it is an automaton that visits only a subset of the states of the original automaton and carries all the information relevant for verification.

We define any infinite string $s \in L^\omega(q)$ originating at state q to be a *fair* string iff for every persistent set Γ at q there is at least one transition σ that occurs in s that is dependent (i.e., not independent) on some

transition in Γ [7, 3]. The following lemma relates a single leap of the leap automaton $Leap(A)$ to the transitions of A.

Lemma 1
Let q be a state common to A and $Leap(A)$. Let s be a string originating at the state q of A that is either finite and leads to a deadlock state or is infinite and fair. Then there exists a finite string $s' = \lambda.\omega$ such that if s is finite $s' \equiv s$ and if it is infinite s has a finite prefix t with $s' \equiv t$ and $q \stackrel{\Theta}{\longrightarrow}_{Leap(A)} q'$ where Θ is the set of all the transitions in the string λ.

Proof of Lemma :
For each persistent set $\Gamma \in \mathcal{G}(q)$ at q let $\gamma(\Gamma)$ be the unique transition in Γ that occurs earliest in the string s. Note that each persistent set must have at least one transition that occurs in the string s either because there is a deadlock state and no transition can remain enabled until then: or because of the fairness condition that a transition cannot remain enabled forever without another transition executing that is dependent on it. Denote the set of all distinct transitions obtained in this way and enabled at q by the symbol Θ'. We describe below a method to construct a subset Θ of Θ' which is a maximal leap set.

Order the elements of Θ' according to the sequence in which they appear earliest in the string s and call these ordered elements $\theta'_1, \theta'_2, \ldots, \theta'_{n'}$. Let $\mathcal{G}_k$ be the set of all persistent sets in $\mathcal{G}(q)$ for which $\Gamma \in \mathcal{G}_k$ iff $\theta'_k \in \Gamma$. Clearly $\cup_{k=1,n'}\mathcal{G}_k = \mathcal{G}(q)$ since every persistent set must be an element of some $\mathcal{G}_k$ by construction. Now let Θ be any subset of Θ' such that :
(1) $\cup_{k\in K}\mathcal{G}_k = \mathcal{G}(q)$ where K denotes the index set of the transitions of Θ' that also remain in Θ :
(2) If $K' \subset K$ then $\cup_{k\in K'}\mathcal{G}_k \subset \mathcal{G}(q)$ where the set inclusions are both strict.

If K is an index set for which (1) above is satisfied we define $rem(K) \subseteq K$ such that for any $j \in rem(K)$, $K \setminus \{j\}$ also satisfies (1). The algorithm to compute a subset Θ can then be expressed as follows :

Maximal Leap Set Algorithm
(I) Set $K_0 ::= \{1, \ldots, n'\}$. $i ::= 0$ and go to (II)
(II) If $rem(K_i) = \emptyset$, stop $K ::= K_i$ is the desired index set for Θ; else go to (III)
(III) Choose any $j \in rem(K_i)$. set $K_{i+1} ::= K_i \setminus \{j\}$ and go to (II).

We claim that Θ given by the algorithm is a maximal leap set. To prove that it is a leap set we first note that for each $k \in K$ there exists at least one persistent set $\Gamma_k \in \mathcal{G}_k$ such that $\Gamma_k \notin \mathcal{G}_u$ for all $u \in K\setminus\{k\}$. For if this is not true then K fails to satisfy the maximality condition given by (2) above since (2) fails with $K' := K \setminus \{k\}$. We can then choose for each θ'_k the special persistent set Γ_k for $k \in K$ which proves the requirement for a leap set since $\theta'_j \notin \Gamma_k$ for any $j \in K \setminus \{k\}$. The fact that it is maximal follows from the fact that $K := K_i$ satisfies (1) when the algorithm terminates.

We now let $\lambda := \theta_1.\theta_2 \ldots .\theta_n$ where $\theta_1, \ldots, \theta_n$ are the elements of the set Θ. Note that if s is finite then $s \equiv \lambda.\omega$ where ω is obtained from s by deleting the earliest instances of each θ_j, $j = 1, \ldots, n$, in it. If s is infinite then choose the finite prefix t of s as the shortest prefix of s that incorporates all the occurences θ_j. Clearly $t \equiv \lambda\omega$ where, this time, ω is the finite string obtained from t by deleting all the instances θ_j in it. In both cases the equivalence relation is an obvious consequence of the definition. This completes the proof. *QED*

Repeated use of Lemma 1 immediately leads to the following result on deadlocks.

Theorem 1
Let A be given automaton and $\{\mathcal{G}(q)\}_{q\in Q}$ a collection of complete persistent sets. A string s originates at the initial state q_0 and terminates at a deadlock state q_d for the automaton A if and only if $s \equiv s_0.\ldots.s_n$ such that $q_0 \stackrel{\Theta_0}{\longrightarrow}_{Leap(A)} q_1 \cdots q_n \stackrel{\Theta_n}{\longrightarrow}_{Leap(A)} q_d$ with Θ_j denoting the set of transitions that occur in the string s_j.

The next theorem shows that every infinite fair string of A can be generated, up to independence equivalence, by the leap automaton $Leap(A)$

Theorem 2
Let A be given automaton and $\{\mathcal{G}(q)\}_{q\in Q}$ a collection of complete persistent sets. The string s is an infinite fair string of A that originates at the initial state q_0 of A if and only if for each integer n there exists a finite prefix t^n of s such that $t^n \equiv s_0.s_1 \ldots s_n s^n$ and $q_0 \stackrel{\Theta_0}{\longrightarrow}_{Leap(A)} q_1 \cdots q_n \stackrel{\Theta_n}{\longrightarrow}_{Leap(A)} q_{n+1}$ where Θ_j denotes the set of transitions that occur in s_j.

Proof of Theorem:
Let n be given and apply Lemma 1 first to generate $s'_0 := s_0.\omega_0$. where $s'_0 \equiv t^0$ with t^0 a finite prefix of s. By lemma we have $q_0 \stackrel{\Theta_0}{\longrightarrow}_{Leap(A)} q_1$. where Θ_0 is the set of events that occur in s_0 and is a maximal leap set. Now assume that construction is complete with $s'_{n-1} := s_0.s_1 \ldots s_{n-1}.\omega_{n-1}$ with $s'_{n-1} \equiv t^{n-1}$ where t^{n-1} is a finite prefix of s and statement of the theorem holds up to $n-1$. Now apply the lemma at the state q_n and s replaced by $\omega_{n-1}.s'$ where s' is defined via the relation $s = t^{n-1}.s'$. If t is the finite prefix of $\omega_{n-1}.s'$ upon application of the lemma, we

choose $t'' = t^{n-1}.s''$ where s'' is the part of s' that is incorporated in the finite prefix t (it could be the empty string). The proof for the converse result is obvious. QED

Remark 3
(1) The most trivial case of the leap automaton is when each set $\mathcal{G}(q)$ is a singleton with the set of all enabled transitions which is complete. Under this restriction the leap automaton and the original automaton coincide. Every leap set Θ is a singleton and no leaping - thus no complexity relief - occurs. If two complete collections are related by $\mathcal{G}(q) \subseteq \mathcal{G}'(q)$ for all $q \in Q$ then the leap automaton $Leap'(A)$ corresponding to the primed collection leaps over the leap automaton $Leap(A)$. This is because every maximal leap set relative to $\mathcal{G}(q)$ at q is a leap set that is not necessarily maximal relative to $\mathcal{G}'(q)$ since the latter has additional persistent sets that may add new transitions to the original leap set. In the same spirit of the above proof every string s of the leap automaton $Leap(A)$ relative to $\mathcal{G}(q)$ can be segmented as $s_1.s_2.\ldots$, where each $s_j = \Theta_1^j.\Theta_2^j \ldots \Theta_n^j$ corresponds to a single transition of the primed leap automaton with $\Theta = \cup_{k=1,n}\Theta_k^j$. This corresponds to the obvious observation that the more refined the collection of persistent sets at a state the greater is the leap distance and the associated complexity relief via by-passing intermediate states and transitions. The trade-off here depends on the burden in computing complete and richer persistent set collections.

(2) $Leap(A)$ can be used for LTL model-checking algorithms instead of A. Given a next-time free LTL formula ϕ, let $vis_\phi \subseteq \Sigma$ be the set of visible transitions where $\sigma \in vis_\phi$ iff there exist two states q, q' such that $q \xrightarrow{\sigma} q'$ and there is an atomic proposition p occuring in ϕ such that the truth value of p at q is not the same as that of p at q'. In other words, execution of the transition σ can change the truth value of an atomic proposition in ϕ. It is shown in [7] that if the dependency relation D satisfies $vis_\phi \times vis_\phi \subseteq D$ and $s \equiv s'$, then s satisfies ϕ iff s' satisfies ϕ. Since under the fairness assumption, $Leap(A)$ can produce an equivalent string for each string A can produce, $Leap(A)$ can be used for next-time free LTL model checking provided that the dependency relation used satisfies $vis_\phi \times vis_\phi \subseteq D$.

(3) We replace the depth-first-search (DFS) used for A with ample sets in [7] by a DFS for the leap automaton $Leap(A)$ taking all transitions into account. If the leap automaton is explicitly modeled, then other algorithms (e.g. the supertrace algorithm) can be used to further reduce complexity.

4 Conclusions

We presented results that show that properties of an automata A to be verified can be reduced verifying its leap automata $Leap(A)$ which, in general, is of much smaller size both in terms of its transition and reachability sets. Our results are presently restricted to fair strings and in that sense more restricted compared to results in [7]. Yet, notwithstanding the restriction to fair strings, our results are more general than their predecessors owing to the definition of maximal leap sets. Currently efforts are on the way to extend the results to the case that entails unfair strings where transitions are classified into silent and visible ones as in [7].

References

[1] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Explosion Problem*. PhD thesis, Universite De Liege, 1994-95.

[2] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.

[3] M.Z. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1:213–228, 1989.

[4] A. Mazurkiewicz. Trace semantics. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets, LNCS 255*, pages 279–324, Bad Honnef, 1987. Springer Verlag.

[5] K. Özdemir and H. Ural. Protocol validation by simultaneous reachability analysis. Accepted for publication in Computer Communications.

[6] K. Özdemir and H. Ural. Deadlock detection in CFSM models via simultaneously executable sets. In *6th Int. Conf. On Communication and Information*, pages 673–688, Peterborough, Ontario, Canada, 1994.

[7] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.

[8] *Functional Specification and Description Language (SDL). CCITT Blue Book, Recommendation Z.100*. Geneva, 1992.

[9] A. Valmari. A stubborn attack on state explosion. In *Computer Aided Verification*, pages 156–165, New Brunswick, NJ, USA, June, 1990.

[10] H. van der Schoot and H. Ural. On improving simultaneous reachability analysis for the efficient verification of deadlock-freedom. Technical Report TR-95-20, Dept. of CSI, University of Ottawa, December 1995.

[11] H. van der Schoot and H. Ural. An improvement in partial-order model-checking with ample sets. Technical Report TR-96-11, Dept. of CSI, University of Ottawa, September 1996. revised in 1997.

EK 11

Reprinted from

INFORMATION AND SOFTWARE TECHNOLOGY

Information and Software Technology 41 (1999) 799–812

Efficient checking sequences for testing finite state machines

K. Inan[a], H. Ural[b,*]

[a]Faculty of Engineering and Natural Sciences, Sabanei University, Istanbul, Turkey
[b]Department of Computer Science, University of Ottawa, Ontario, Canada

ELSEVIER

Co-Editors

Michael Dyer
Suite 780,
7201 Wisconsin Avenue,
Bethesda, MD 20814,
USA
E-mail: mdyer@access.digex.net

Professor Martin Shepperd
Department of Computing,
Bournemouth University,
Poole House, Talbot Campus,
Fern Barrow, Poole, Dorset BH12 5BB, UK
E-mail: mshepper@bournemouth.ac.uk

International Editorial Board

D.J. Andrews
Head of School of Information Systems,
Nene College of Higher Education,
Park Campus, Boughton Green Road,
Northampton NN2 7AL, UK

Dr B.W. Boehm
Centre for Software Engineering,
University of Southern California,
Los Angeles, CA 90089-0781, USA

Professor M. Broy
Institut für Informatik der Technischen,
Universität München, Postfach 20 24 20,
80290 München, Germany

I.J. Campbell
GIE Emeraude, 38 Boulevard H-Sellier,
92154 Suresnes-Cedex, France

D.N. Card
Software Productivity Consortium,
2214 Roch Hill Road, Herndon,
VA 20170-4227, USA

Professor D.N. Chorafas
Domaine Valmer, 06360 Saint Laurent,
d'Eze, Alpes-Maritimes, France

Professor B.J. Garner
Deakin University, Geelong,
Victoria 3217, Australia

Professor P.A.V. Hall
Computing Department, Open
University, Walton Hall, Milton Keynes,
MK7 6AA, UK

Dr P. Hitchcock
School of Computer Engineering,
Technical University of Nova Scotia,
PO Box 1000, Halifax,
Nova Scotia, Canada B3J 2X4

Professor D. Ince
Faculty of Mathematics, Open
University, Walton Hall,
Milton Keynes
MK7 6AA, UK

Professor M. Jackson
School of Computing and Information
Technology,
University of Wolverhampton,
Wulfruna Street, Wolverhampton
WV1 1SB, UK

Dr T. Matsubara
1-9-6 Fujimigaoka, Ninomiya,
Nakagun, Kanagawa 259-01,
Japan

Monika Müllerburg
GMD, Postfach 1240,
Schloss Birlinghoven,
D-5205 Sankt Augustin 1, Germany

Professor C. Rolland
Université de Paris 1,
Sorbonne, 17 rue de la Sorbonne,
75231 Paris Cedex 05, France

Dr Marc Roper
Department of Computer Science,
University of Strathclyde,
Livingstone Tower, Richmond Street,
Glasgow G1 1XH, UK

Dr T. Takeshita
School of Business Administration
and Information Science,
Chubu University,
Kasugai City, Japan

Professor T. Tamai
College of Arts and Sciences,
University of Tokyo, 3-8-1 Komaba,
Meguro-ku, Tokyo 153, Japan

J.J. van Amstel
Philips Research Laboratories,
Information and Software Technology,
Professor Hostlaan 4, 5656 AA
Eindhoven, The Netherlands

Professor Claes Wohlin
Department of Communication Systems,
Lund Institute of Technology,
Lund University
Box 118, S-22100 Lund,
Sweden

J.B. Wordsworth
IBM United Kingdom Laboratories Ltd,
Hursley Park, Winchester,
Hampshire SO21 2JN, UK

J. Verner
College of Information Sciences and
Technology,
University of Drexel,
Rush Building,
33rd and Market Street,
Philadelphia, PA 19104, USA

Information and Software Technology is an international technical journal which covers all aspects of software development and information processing, from state-of-the-art research, through software development and implementation, to information systems management. The journal gives equal emphasis to the theories of software engineering and the application of information technology within organizations.

Papers published in the journal are drawn from current developments in areas such as: empirical and experimental analyses, software metrics, software processes and development methods, project management, quality control and standards, object orientation, concurrency, human factors, testing, implementation techniques, database design and information systems, to provide a total view of informations systems technology.

Contributions Those wishing to submit full-length papers, tutorials or review papers should send four copies to either of the Co-Editors. Contributors should refer to the Notes for Authors printed in this issue of the journal. These are also available from the publishers.

ELSEVIER

Information and Software Technology 41 (1999) 799–812

INFORMATION AND SOFTWARE TECHNOLOGY

Efficient checking sequences for testing finite state machines

K. Inan[a], H. Ural[b,*]

[a]Faculty of Engineering and Natural Sciences, Sabanei University, Istanbul, Turkey
[b]Department of Computer Science, University of Ottawa, Ontario, Canada

Abstract

A general model for constructing minimal length checking sequences is proposed. The model is based on the characteristics of checking sequences and sets of state identification and verification sequences. Some existing methods are shown to be special cases of the proposed model. The minimality of the resulting checking sequences is discussed and heuristic algorithms for the construction of minimal length checking sequences given. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Checking sequence; Mealy automaton; Distinguishing suite; State identification and verification

1. Introduction

Testing the implementations of systems with respect to their specifications is a crucial activity to ensure their correct functioning. The specification of a system in diverse areas such as switching, telephone, communication protocols, lexical analysis, pattern matching, and machine learning can be modeled by a Mealy automaton. Foundations of testing an implementation of a system, modeled as an automaton, can be found in a sequential circuit and switching system testing literature [1–5], where determining, under certain assumptions, whether any given "black box" implementation M' of a Mealy automaton M is functioning correctly is referred to as a *fault detection experiment*. This experiment consists of applying an input sequence (derived from M) to M', observing the actual output sequence produced by M' in response to the application of the input sequence, and comparing the actual output sequence to the expected output sequence (derived from M). The applied input sequence is a *checking sequence* relative to M.

Given M, the construction of a checking sequence from M that determines whether any given implementation M' is a correct implementation of M is sought in practice by considering some basic assumptions. The assumptions, commonly made in the literature [1–14], are that M is deterministic, minimal, completely specified and represented by a strongly connected directed graph, that M' is deterministic and has the same set of inputs and outputs as M, and that faults do not increase the number of states in M' but may alter the output and destination of transitions in M'. Further, the construction of a checking sequence must deal with the "black box" nature of a given implementation M' of M which allows only limited controllability and observability of M', i.e. in order to deduce that every state transition of M is implemented correctly by M', to verify the state transition from state s_j to s_k under input x, (a) before the application of x, M' must be transferred to the state recognized as s_j, (b) the output produced by M' in response to the application of x must be as specified in M, (c) the state reached by M' after the application of x must be recognized as s_k.

Hence, a crucial part of testing the correct implementation of each transition is recognizing the starting and terminating states of the transition. The recognition of a state of an automaton, M, can be achieved by a distinguishing sequence, unique input–output sequence, or a set of characterizing sequences. Based on these sequences, a variety of methods for the construction of checking sequences have been proposed in the literature. Some of these methods make a restrictive assumption of reliable reset feature, which requires an implementation of a given automaton to correctly reset to the initial state from any state under a specific reset input [6,7,9,12]. The methods that do not make the restrictive assumption also do not make an explicit attempt to obtain a minimal length checking sequence [10,14]. The minimality of the length of an input sequence to be used for testing purposes is studied outside the context of checking sequences [8,15–21]. These studies call the resulting minimal length input sequence a test sequence due to the fact that not all the components of a checking

* Corresponding author.
E-mail addresses: inan@sabanciuniv.edu.tr (K. Inan); ural@csi.uottawa.ca (H. Ural)

0950-5849/99/$ - see front matter © 1999 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(99)00038-5

sequence are present in the input sequence. Some of these studies concentrated on the analysis of the effects of sub-sequences on the length of the input sequence [17,18,21].

In this work, based on the characteristics of checking sequences, a general model for constructing minimal length checking sequences is proposed. In Section 2, related terminology is introduced. In Section 3, characteristics of state identification and verification sequences are discussed. In Section 4, a general model for formulating minimal length checking sequences is presented and a proof for the resulting sequences to be checking sequences is given. This section also shows that the previous methods are special cases for the proposed model. In Section 5, minimality of the resulting checking sequences based on the concept of overlapping is discussed and a heuristic algorithm for the construction of minimal length checking sequences is given. Section 6 gives the concluding remarks.

2. Preliminaries

We assume that we are given a Mealy automaton $M = (Q, \Sigma_i, \Sigma_o, \delta, \lambda, q_0)$, where Q is the finite state set, Σ_i and Σ_o, the finite input and output alphabets, $\delta : Q \times \Sigma_i \to Q$, the transition function, $\lambda : Q \times \Sigma_i \to \Sigma_o$ the output function and q_0 the initial state. We also assume that M is *deterministic* and *complete*, i.e. the transition function, δ, and the output function, λ, are both total functions. We let without ambiguity δ and λ also denote their reflexive and transitive closures. Thus, δ and λ are extended to the functions $\delta : Q \times \Sigma_i^* \to Q$ and $\lambda : Q \times \Sigma_i^* \to \Sigma_o^*$.

In terms of the above definition we can view an automaton M as an edge labeled multi-graph $G = (V,E)$, where the set V of vertices and the set E of edges correspond to the set Q of states and the set of transitions of M, respectively. We assume that M is represented by a strongly connected digraph. Each edge $e = (v_j, v_k; x/y) \in E$ is a state transition from state $s_j \in Q$ to state $s_k \in Q$ with input $x \in \Sigma_i$ and output $y \in \Sigma_o$, where v_j is the head of e denoted by head(e), v_k is the tail of e denoted by tail(e), the input/output (or i/o) pair x/y is the *label* of e denoted by $l(e)$. A *transfer sequence* in G from vertex v_i to vertex v_j is the projection of the label of a path from v_i to v_j on Σ_i. An automaton M has the reset feature if there is an input r such that $\delta(q, r) = q_0$ and $\lambda(q, r) = null$ for each $q \in Q$.

Next we first present a notation to be used in subsequent sections in characterizing the paths in a digraph $G = (V,E)$, representing a given Mealy automaton M. A path p in G is a sequence of edges given as $p = e_1.e_2, \ldots, e_n$, $n \geq 1$. A subpath p' of p is a sequence of edges given by: $e_k.e_{k+1}, \ldots, e_m$, where $1 \leq k \leq m \leq n$. For a path p in G, the label of p denoted by $l(p)$ stands for the concatenation of labels of the edges in p. Let head(p) and tail(p) represent the head(e_1) and tail(e_n) of the path $p = e_i, \ldots, e_n$. Also, let $l_i(p)$ and $l_o(p)$ denote the projections of the label sequences on Σ_i and Σ_o, respectively. For any path p, let p^j denote the jth element of the vertex sequence corresponding to the path p, and $l(p)^j = (l_i(p)^j / l_o(p)^j)$ is the label of the jth edge in the path. The *cost* (or *length*) of an edge is the number of i/o pairs in the label of the edge. The *cost* (or *length*) of a path p is the sum of the lengths of edges included in p. A path with the minimum length among all paths from v_s to v_t is called a *shortest path* from v_s to v_t. The concatenation of two sequences (or paths) p and p' is denoted by $p.p'$. For the rest of the paper we assume that the reader is familiar with the graph theoretic terminology [22–24].

Next we develop concepts related to the equivalence of automata. For this we first define equivalence of two states q, q' of an automaton M by considering $q \equiv q'$ iff $\lambda(q, s) = \lambda(q', s)$ for all $s \in \Sigma_i^*$. We can extend this definition of equivalence to the states of two automata. Hence, let $M' = (Q', \Sigma_i, \Sigma_o, \delta', \lambda', q_0')$ be another automaton over identical input and output alphabets. Assume without loss of generality that $Q \cap Q' = \emptyset$. If $q \in Q$ and $q' \in Q'$, then let $q \equiv q'$ iff $\lambda(q, s) = \lambda'(q', s)$ for all $s \in \Sigma_i^*$. We say that M is equivalent to M', denoted by $M \equiv M'$, iff $q_0 \equiv q_0'$. We say that M' is *homomorphic* to M, if there exists some function (homomorphism) $f : Q \to Q'$ that preserves the transitions and outputs in the sense that $\delta'(f(q), \sigma) = f(\delta(q, \sigma))$ and $\lambda'(f(q), \sigma) = \lambda(q, \sigma)$, for every $q \in Q$, $q' \in Q'$, and $\sigma \in \Sigma_i$. M' is said to be isomorphic to M, if it is homomorphic where the homomorphism f is a bijection. If M and M' are isomorphic and $q_0' = f(q_0)$, then $M \equiv M'$. M is said to be a *minimal* representation if distinct states in M are not equivalent. If M is not a minimal state machine then there is a quotient machine $M/\equiv$ obtained from M which is both minimal and equivalent to M. It is possible to define equivalence of states—and hence, automata—in a restricted manner. Let $I \subseteq \Sigma_i^*$ be a given set of input sequences, we then denote $q \stackrel{I}{\equiv} q'$ iff $\lambda(q, s) = \lambda(q', s)$ for all $s \in I$. Extension to $M \stackrel{I}{\equiv} M'$ follows the obvious logic.

Let M be a given automaton and let Φ_M be a set of Mealy automata that contains M. Then, a checking sequence s relative to M and the set Φ_M is an element of Σ_i^* such that

$$\forall M' \in \Phi_M \left(M \equiv M' \Leftrightarrow M \stackrel{\{s\}}{\equiv} M' \right). \tag{1}$$

In the following sections, we shall design sequences s in order to prove that a black box automaton M', subject to certain constraints, represented by the set Φ_M responds to the sequence s in precisely the same manner as a given white box automaton M, if and only if it is equivalent to M. Clearly, according to the above definition this is true if and only if such a sequence s is a checking sequence relative to M and Φ_M. In general, we assume that elements of the set

Φ_M are deterministic[1] automata over the same input and output alphabets as M. Later we shall assume that the number of states of elements of Φ_M are uniformly bounded by some integer.

3. A model for state identification and state verification

In general, the construction of a checking sequence consists of the following three distinct stages:

1. Construction of path segments for *state identification*;
2. Construction of paths for *transition* and *state verification*;
3. Integrating results of the previous stages to compute a low cost (optimal relative to the elements of (1) and (2) above) checking sequence.

Given an implementation M' of an automaton M, the first stage involves deciding those input segments by which we can distinguish $|Q|$ distinct states of the implementation (black box) M'. The second stage involves verification of each transition (edge) of M in M'. This stage naturally involves the recognition of head and tail of the transitions in question as states of M. It will be shown later that the input segments that identify the states and those that verify them are related, but they may not be identical. Finally, the third stage involves defining a general problem of optimal integration of the components given in the identification and verification stages.

In this section we present a general structure to construct the necessary input sequences for state recognition (i.e. state identification and state verification). In Section 4, we formulate the integration optimization problem and discuss the special cases corresponding to various assumptions made in the literature.

The key concept under state identification is given by the following definition.

Definition 1. Let $U(q)$ denote a set of input sequences in Σ_i^* for each state $q \in Q$ of a given M. A collection $U := \{U(q)\}_{q \in Q}$ is called a distinguishing suite for M if the following condition holds: for all $q, q' \in Q$ with $q \neq q'$, there exists $u \in U(q)$ and $u' \in U(q')$ and $p \in \Sigma_i^*$, such that $p \leq u$, $p \leq u'$ and $\lambda(q,p) \neq \lambda(q',p)$ where '$\leq$' stands for the string prefix relation.

In order to ensure that a distinguishing suite has no redundancies, we first define a partial order on the set of all *distinguishing suites* for M and derive a concept of efficiency relative to this partial order.

[1] It is not necessary to assume that elements of Φ_M are *complete* and *minimal* Mealy automata as is sometimes assumed in the literature. Completeness is an indirect consequence of the checking experiment itself and if the automaton under test is not minimal then what we identify is the minimal state equivalent version.

Definition 2. Let I and I' be any two sets of input sequences in Σ_i^* of a given M. We say $I \preceq_s I'$ iff for each $i \in I$, there exists $i' \in I'$, such that $i \leq i'$. Let U, U' denote two distinguishing suites for M. We define a partial order by letting $U \preceq U'$ iff for each $q \in Q$, $U(q) \preceq_s U'(q)$. A distinguishing suite, U, is called efficient if it is minimal relative to '$\preceq$;', i.e. if U' is another distinguishing suite and $U' \preceq U$ then $U' = U$.

Remark 1. (1) Let DS denote a distinguishing sequence for M, then $U(q) := \{\text{DS}\}$ for all $q \in Q$ and the associated U is a distinguishing suite. In order to make this into an efficient distinguishing suite we let for each $q \in Q$, $U(q) := \{\text{DS}(q)\}$ where $\text{DS}(q)$ denotes the shortest prefix of DS which distinguishes q from all the other states. Note that if DS itself is minimal (i.e. it does not have a prefix that is a distinguishing sequence), then at least for one $q \in Q$ we must have $\text{DS}(q) = \text{DS}$. Also, note that for a given DS the efficient distinguishing suite is unique.

(2) We first define a set of *characterizing sequences* (usually referred to as a W-set [7]) as follows:

Definition 3. A minimal set of input sequences $W := \{w_1, w_2, \ldots, w_r\}$ is called a W-set of a given M if it satisfies the property that for every pair $q, q' \in Q$ with $q \neq q'$ there exists a characterizing sequence $w_i \in W$ such that $\lambda(q, w_i) \neq \lambda(q', w_i)$. By minimality, it is meant that no proper subset of W enjoys the property expressed above.

It can be seen that if $U(q) := W$ for all $q \in Q$, then the associated U is a distinguishing suite. It is also intuitively clear that U is not necessarily efficient. In order to obtain an efficient version of U we can first generate different copies of W, say $W(q)$, for each q by trimming or removing elements—which is a special case where the prefix is the empty string!—of W to obtain strict prefixes. Unfortunately, unlike the previous case of a single DS, efficient solutions are abundant as we demonstrate and elaborate below.

In order to pursue the trimming procedure we first arbitrarily order the states in Q as $q_1, \ldots, q_n$. In order to compute U_1 we trim elements in W to obtain a $W(q_1)$ in some arbitrary way such that if U_1 is obtained from U by only replacing $U(q_1)$ by $W(q_1)$, then U_1 is a one-minimal distinguishing suite in the sense that if U' is also a distinguishing suite where $U' \preceq U_1$ and $U'(q_j) = U_1(q_j)$ for $j = 2, \ldots, n$ then $U' = U_1$. Now assume (inductively) we have computed a distinguishing suite U_k, where $U_k(q_m) = U(q_m) = W$ for $m > k$ and $U_k(q_m) = W(q_m)$ for $1 \leq m \leq k$. Also assume that U_k is k-minimal, exactly in the above sense: that if U' is also a distinguishing suite where $U' \preceq U_k$ and $U'(q_j) = U_k(q_j)$ for $j = k+1, \ldots, n$, then $U' = U_k$.

We then select $W(q_{k+1})$ by arbitrarily trimming the elements of W such that if U_{k+1} is obtained from U_k simply by replacing $U_k(q_{k+1}) = W$ by $W(q_{k+1})$ then it is a $k+1$-minimal distinguishing suite. The desired result is given by

U_n which is an n-minimal, or equivalently, an efficient distinguishing suite. Note that in the above construction there are basically two degrees of freedom: the ordering of the states and the manner in which $U_k(q_{k+1}) = W$ is replaced by $U_{k+1}(q_{k+1}) = W(q_{k+1})$. This clearly implies that there is a multiplicity of distinguishing suites that are efficient in a non-comparable manner, save for-somewhat artificial-cost constraints that can be imposed to compute an optimal solution among these efficient solutions.

(3) Now let $u(q)$ denote the projection of a *Unique Input–Output Sequence* (UIO) sequence [11] for state $q \in Q$ on Σ_i. As a special case for the W-set above we can take the collection of UIO sequences given by $W = \{u(q)\}_{q \in Q}$. We pursue this special case to illustrate what the efficient distinguishing suite looks like for a particular choice of state ordering.

The set $W(q_1)$ is a singleton consisting of $u(q_1)$ only. The set $W(q_2)$ consists of two elements, namely $u_2(q_1)$ and $u(q_2)$ where $u_2(q_1)$ is the shortest prefix of $u(q_1)$ that distinguishes the states q_1 and q_2. Generalizing, we see that $W(q_k)$ has k elements with one element $u(q_k)$ and the rest $\{u_k(q_j)\}_{j<k}$ where $u_k(q_j)$ is the shortest prefix of $u(q_j)$ that distinguishes states q_k and q_j. Note that the definitions of $u_k(q_j)$ that appear in this construction are independent of the selected order of states by which the final efficient suite is built.

Now let us pursue the analysis one step further. Define a distinguishing suite U by letting

$$U(q_k) := \{u(q_k)\} \bigcup_{j \neq k} \{u_k(q_j)\}.$$

Also, let U_j denote any of the distinguishing suites, constructed as above, that starts with the state q_j (the order for the rest of the states is irrelevant). We then have the following structure: each distinguishing suite U_j is efficient and in particular has a single input corresponding to state q_j and satisfies the relation $U_j \preceq U$. The application of the jth component of U_j consisting of $u(q_j)$ is sufficient in the *verification* of the state q_j provided that all the elements of $U(q_k)$ are applied at each state q_k at the state *identification* stage. This is because $u(q_j)$ distinguishes the state q_j from any other state q_k, as by assumption $u_k(q_j) \leq u(q_j)$ has been applied to the state q_k at the state *identification* stage to distinguish it from state q_j. This coincides with the usual practice [7,12] except that in the usual case the untrimmed input $u(q_j)$ is used instead of the shortest prefix $u_k(q_j)$.

(4) The above idea can be generalized to the case where a UIO does not necessarily exist for each state q (See Propositions 2 and 3 in Section 4.1).

4. Construction of checking sequences

Before we describe the useful segments to build the checking sequences, we first describe concepts and methods related to identifying the states of a black box by output observations. Let $\mathscr{P}$ denote the set of all *partitions* of the state set Q. We define the function $\Delta : \Sigma_i^* \rightarrow \mathscr{P}$ as follows:

$p \in \Delta(s)$ iff $\exists u \in \Sigma_o^*$ such that

$x \in p \Leftrightarrow \lambda(x, s) = u.$

Note that if $s < s'$ (i.e. s is a prefix of s'), then $\Delta(s')$ refines $\Delta(s)$. The following definitions are relative to a given Mealy automaton M and the function Δ derived from it.

A sequence $s \in \Sigma_i^*$ is called a *Distinguishing Sequence* (DS) iff every element of $\Delta(s)$ is a singleton set. It is known that a DS may not exist for every minimal automaton [1], and determining the existence of a DS is PSPACE-complete [25]. The sequence $s \in \Sigma_i^*$ is called the projection of a UIO on Σ_i corresponding to a state $q \in Q$ iff $\{q\} \in \Delta(s)$. Note that not every state of a minimal automaton has a UIO [11]. The sequence $s \in \Sigma_i^*$ is called a *Characterizing Sequence* (CS) corresponding to a pair of states $p, q \in Q$ iff $\lambda(p, s) \neq \lambda(q, s)$, or equivalently no element of $\Delta(s)$ contains both p and q. It is known that a pair of states p, $q \in Q$ of a minimal automaton can always be distinguished by a sequence of at most $|Q| - 1$ inputs [1].

4.1. Forming elements of checking sequences

The treatment below covers the general case: a DS may not exist for M and possible number of states of each element of Φ_M is bounded by $|Q| + \varepsilon$, where ε is a given fixed integer. The formulation allows for a smooth passage first to the special case where $\varepsilon = 0$—the usual assumption made in the literature—and then to the special case where it is further assumed that a DS exists. In what follows, the terminology may not be the usual one in the literature, hence we caution the reader to consider the underlying reasoning carefully. Most of the definitions involve a path p in the digraph G representing a given automaton M (in short G of a given M) and nodes on this path. We imply the distinction between the formal reference to this node, such as head(p) for some path p, and the corresponding state. Hence, when we refer to a set of nodes on a path the corresponding states of M (or vertices of G) can all be equal to each other as in the *coherency* definition below.

Whenever a property of a state is suggested by the terminology used in the definitions, what usually is meant is not a property of the states of the given automaton M corresponding to the nodes on the path, but the corresponding states of an implementation $M' \in \Phi_M$ of M when the same input derived from the path p, namely $l_i(p)$, is applied to M'. More precisely, when we say that a node x on a path p in M given by $x = \delta(q_0, s)$, where s is a prefix of $l_i(p)$, has a certain property we actually refer to the corresponding state x' of M' given by $x' = \delta'(q'_0, s)$. An implicit understanding behind the concepts is that the outputs of M and M' are identical when subject to the same input. Only when this assumption holds, we have to prove that $M \equiv M'$, as otherwise the test fails and the verdict is negative, i.e. M' is a faulty implementation of M.

A key concept lies in the recognition of some nodes on a

path as having an identical, but an unknown common value (i.e. a state of the automaton M) during an experiment. This concept is captured in the definition of *coherent sets* below.

Definition 4. Let p be a given path in G of a given M that starts at q_o and let $\{x_j\}_{j\in J}$ be a set of nodes on the path p which are considered as having the same value $x = x_j = \delta(q_0, s_j)$, where each s_j is a prefix of $l_i(p)$. The collection $\{x_j\}_{j\in J}$ is said to be a coherent set of nodes iff the corresponding states of an $M' \in \Phi_M$, given by $x'_j := \delta'(q'_0, s_j), j \in J$ are also recognized as having a single value in Q' whenever $M \overset{l_i(p)}{\equiv} M'$.

The proofs of the following three propositions are straightforward and omitted.

Proposition 1. Let p be a given path in G of a given M where coherent sets are subsets of nodes on p.

(a) Every subset of a coherent set is coherent.

(b) Suppose A and B are coherent sets with the same common value, then $A \cup B$ is also a coherent set with the same common value.

(c) If $\{P_j\}_{j\in J}$ is a set of paths on p and $\{\text{head}(p_j)\}_{j\in J}$ is a coherent set, then the set $\{\text{tail}(p_j)\}_{j\in J}$ is also a coherent set provided $l(p_j)$ is a constant label sequence independent of $j \in J$.

Using the concept of coherent sets, the relation between the state identification and state verification stages can be easily highlighted. This relation is explained by the following two propositions.

Proposition 2. Let U_j and U be two given distinguishing suites with $U_j \preceq U$. Let p be any path in G of a given M that starts in q_o, and assume that $M' \overset{l_i(p)}{\equiv} M$ and $|Q'| \le |Q|$. Also assume that for each $q \in Q$ and $u \in U(q)$, p has a subpath $p(u,q)$ starting at state q and $l_i(p(u,q)) = u$ and for each $v \in U_j(q_j)$ there is a subpath of $p_j(v)$ starting at state q_j such that $l_i(p_j(v)) = v$. Under these conditions if the sets $\{\text{head}(p(u,q_j))|u \in U(q_j)\}$ and $\{\text{head}(p_j(v))|v \in U_j(q_j)\}$ are coherent, so is the set $\{\text{head}(p(u,q_j))|u \in U(q_j)\} \cup \{\text{head}(p_j(v))|v \in U_j(q_j)\}$.

Now let a W-set of M be assumed to be given and let $W(q_j) \preceq W$ be a minimal set that distinguishes q_j from every other state $q \neq q_j$. We now construct a not necessarily efficient, distinguishing suite U for M as follows:

(i) $W(q_j) \subseteq U(q_j)$,

(ii) For each $k \neq j$, choose any $w \in W(q_k)$ that will distinguish q_k from q_j. Then if w' is the shortest prefix of w that still distinguishes q_k from q_j, then let $w' \in U(q_j)$.

Proposition 3. Under the choice of $U(q_j)$ above the set U is a distingnishing suite, and for each q_i there exists an efficient distinguishing suite U_j such that $U_j(q_j) = W(q_j)$ and $U_j \le U$.

Remark 2. In the same spirit as in (3) of Remark 1, as long as all the inputs in $U(q_j)$ are applied to states $q_j, j = 1, \ldots, n$, at the state identification stage, it is enough to apply the inputs in $U_j(q_j) = W(q_j)$ to state q_j at the state verification stage. This generalizes the distinction between the identification and verification stages of the states of the unknown automaton M' in a precise manner.

For the case where a DS exists for a given M, coherent sets on a path p can easily be identified. If $s_j := s'_j.DS.s$ are prefixes of $l_i(p)$ for $j \in J$, then the nodes $x_j = \delta(q_0, s_j)$ form a coherent set, provided the outputs $\lambda(y_j, DS)$are identical for $j \in J$ where $y_j := \delta(q_0, s'_j)$. Note that $\{y_j\}$ is also a coherent set.

For the case where a DS does not exist, identifying coherent sets is more prominent. For this we can employ a W-set. The difficulty of identifying states arises from the need to apply more than one sequence in a W-set at a single state without any reset feature to repeat the experiment. For this, one uses additional concepts originally suggested in Ref. [5] and generalized in Ref. [10]. The key concept here is that of a *locating sequence*.

Recall that $\varepsilon \ge 0$ is differential bound, taken with respect to the number of states of M, on the number of states of $M' \in \Phi_M$. We shall define a locating sequence and derive associated results for the general case of $\varepsilon > 0$. For this we shall need the following definitions and results. The W-set defined for the automaton M is extended to another finite set of input sequences called the Z-set given by

$$Z := W \cup \Sigma_i.W \cup \ldots \cup (\Sigma_i)^{\varepsilon}.W, \quad (2)$$

where for any language L, L^k denotes k-fold language self-concatenation. For the sake of analysis, we define $Z(k)$by

$$Z(k) := W \cup \Sigma_i.W \cup \cdots \cup (\Sigma_i)^k.W. \quad (3)$$

Note that $Z(0) = W$ and $Z(\varepsilon) = Z$. Also note that

$$|Z(k)| = \sum_{j=0,k} (|W| * |\Sigma_i|^j) = |W| * \frac{|\Sigma_i|^{k+1} - 1}{|\Sigma_i| - 1}.$$

We define a partition function $\Delta : N \to \mathcal{P}$ relative to the automaton M' in Φ_M, where N denotes the set of nonnegative integers and the set $\mathcal{P}$ denotes the set of all partitions defined in Q'. Δ is defined by considering

$$p \in \Delta(k) \Leftrightarrow \lambda'(x,z) = \lambda'(y,z), \ \forall x, y \in p, \ \forall z \in Z(k).$$

As $Z(k) \subseteq Z(k+1) = \Sigma_i.Z(k) \cup W$, it follows that the partition $\Delta(k+1)$ refines $\Delta(k)$.

Lemma 1. Suppose that $M \overset{l_i(p)}{\equiv} M'$, where p is any path in G of a given M with the property that there exists subpaths $\{p_{jm}\}_{m=1,|W|;j=1,|Q|}$, such that $\text{head}(p_{jm}) = x_j$ for each m and

$l_i(p_{jm}) = w_m$, where x_j denote elements of Q. Under these conditions $\Delta(0) \geq |Q|$.

Proof. The responses of M at each x_j is distinguished at least by one element w_m in W by definition and by $l_i(p)$-equivalence, identical responses identify |Q| distinct states in M', hence the partition induced by $W = Z(0)$ satisfies $\Delta(0) \geq |Q|$. □

Lemma 2. If $\Delta(k)$ is not atomic (i.e. elements are singleton sets) then $|\Delta(k+1)| \geq |\Delta(k)| + 1$.

Proof. As $\Delta(k)$ is not atomic, there exists $p \in \Delta(k)$ and $x,y \in p$ such that $x \neq y$. Clearly, there cannot exist σ such that $\lambda'(x,\sigma) \neq \lambda'(y,\sigma)$ since this would violate the definition of p. On the contrary, if for any σ, $\delta'(x,\sigma) = \delta'(y,\sigma)$then x and y can never be distinguished, and as $x \neq y$ this would violate the minimality assumption of automaton M'. □

Therefore, for all σ the corresponding outputs are the same and there must exist elements $p'(\sigma), p''(\sigma) \in \Delta(k)$ such that $x' := \delta'(x,\sigma) \in p'$ and $y' := \delta'(y,\sigma) \in p''$ with $x' \neq y'$. If $p' \neq p''$ then the lemma follows, as x', y' can be distinguished by some $z \in Z(k)$ which implies that x and y can be distinguished by $\sigma.z \in Z(k+1)$ and therefore the set p splits into $\Delta(k+1)$ which strictly refines $\Delta(k)$.

The only other possibility left is when $p' = p''$. By applying the same argument applied to $x, y \in p$ to $x', y' \in p'$, we either arrive at a contradiction or prove the lemma as demonstrated above; or as there are a finite number of partitions and states, we enter a cycle where a previous pair of states and a partition repeats. This again contradicts the minimality of M' as none of the pair of states appearing in this cycle can be distinguished.

We now define a locating sequence based on a Z-set defined above.

Definition 5. A locating sequence denoted by $L_{qq'}$ is an input sequence in Σ_i^* that satisfies the following property:

Let p be a path in G of a given M that starts at q_o and $\{p_j\}_{j \in J}$ be any set of subpaths of p with the properties: head$(p_j) = q$ and $l_i(p_j) = L_{qq'}$, where $q' := \delta(q, L_{qq'})$. If s_j is the input projection of the label of a prefix path of p that terminates at tail(p_j), then the collection of nodes $\{x_j | x_j := \delta(q_0, s_j)\}$, is a coherent set with a common value q'.

The following proposition is on the existence of locating sequences. The constructive proof of this proposition and the associated complexity results on the size of the sequence is given in the Appendix.

Proposition 4. For any $q, q' \in Q$ of a given M there exists a locating sequence $L_{qq'}$. If $L_{vv'}$ is another sequence obtained from $L_{qq'}$ by attaching a prefix and a postfix to it, derived from a path prefix and postfix of p, respectively, then it is also a locating sequence.

The key property of locating sequences is their ability to generate coherent sets. The following definitions generalize an original idea given in Ref. [2] and used in Ref. [13] in the context of distinguishing sequences, notwithstanding differences induced by the generalizations and the concept of coherency.

Definition 6. Let p be a path in G of a given M that starts at q_o.

(a) Let $\{p_j\}_{j=1,|Z(\varepsilon)|}$ be a collection of subpaths of p, then the set of nodes $\{\text{head}(p_j)\}_{j=1,|Z(\varepsilon)|}$ is said to be z-recognized in p if it is a coherent set and $l_i(p_j) = z_j$.

(b) A node x on p is said to be t-recognized in p if there exists a set of subpaths $\{p_j\}_{j=1,|Z(\varepsilon)|}$ such that $\{\text{head}(p_j)\}_{j=1,|Z(\varepsilon)|}$ is a z-recognized set and the set $\{\{x\} \cup \{\text{head}(p_j)\}_{j=1,|Z(\varepsilon)|}\}$ is a coherent set.

The common value of a coherent set is identified by the responses of the elements of the coherent set to the elements of the special subset of the Z-set, namely the W-set.

The next definition is on the verification of an edge with the Z-set.

Definition 7. An edge e of G of a given M is said to be verified along a path p in G that starts at q_o if there exists a subpath p' of p such that head(p') is t-recognized in p as head(e), $l(p') = l(e)$ and tail(p') is t-recognized in p as tail(e).

The following proposition shows how t-recognition proliferates through edge verification.

Proposition 5. Let p be a path in G of a given M that starts at q_o and let p' be a subpath of p such that all edges of G that correspond to all edges of the subpath p' are verified in p. Under these conditions if head(p') is t-recognized in p then tail(p') is t-recognized in p.

Proof. In what follows, all the t- and z-recognitions are with respect to path p. Let e' be the final edge in the subpath p'. By a standard induction argument on the length of the subpath p', we assume that head(e') is t-recognized and it has to be proved that tail(e') is t-recognized. As the edge e of G corresponding to e' in p' is verified there exists a subpath p_1 of p such that head(p_1) is t-recognized as head(e), $l(p_1) = l(e) = l(e')$ and tail(p_1) is t-recognized as tail(e). As head(e') is t-recognized there are subpaths $\{v_j'\}_{j=1,|Z(\varepsilon)|}$ of p such that $\{\text{head}(v_j')\}_{j=1,|Z(\varepsilon)|}$ is z-recognized and $\{\{\text{head}(e')\} \cup \{\text{head}(v_j')\}_{j=1,|Z(\varepsilon)|}\}$ is a coherent set. On the contrary, as head(p_1) is t-recognized as head(e) there exists subpaths $\{p_{2j}\}_{j=1,|Z(\varepsilon)|}$ such that $\{\text{head}(p_{2j})\}_{j=1,|Z(\varepsilon)|}$ is z-recognized and $\{\{\text{head}(e')\} \cup \{\text{head}(p_{2j})_{j=1,|Z(\varepsilon)|}\}$ is also a coherent set. However, as the common value of both the coherent

sets equals head(e), we must have {head(e'), head(p_1)} also a coherent set by (a) and (b) of Proposition 1. This implies, this time by (c) of Proposition 1 that {tail(e'), tail(p_1)} is also a coherent set as $l(p_1) = l(e') = l(e)$. □

It remains to be proved, using the fact that tail(p_1) is t-recognized, that tail(e') is t-recognized. The latter implies the existence of subpaths $\{p_{3j}\}_{j=1,|Z(\varepsilon)|}$ such that $\{\{\text{tail}(p_1)\} \cup \{\text{head}(p_{3j})\}_{j=1,|Z(\varepsilon)|}\}$ is a coherent set and $\{\text{head}(p_{3j})\}_{j=1,|Z(\varepsilon)|}$ is z-recognized in p, which in turn, implies that $\{\{\text{tail}(e')\} \cup \{\text{head}\{p_{3j}\}_{j=1,|Z(\varepsilon)|}\}$ is a coherent set where $\{\text{head}(p_{3j})\}_{j=1,|Z(\varepsilon)|}$ is z-recognized. This proves that tail(e') is t-recognized and the proof is complete.

The main result that relates these concepts to a set of sufficient conditions for the input projection of the label of a path to be a checking sequence, is given by the following theorem.

Theorem 1. Let p be a path in G of a given M that starts at q_o. Then the sequence $l_i(p)$ is a checking sequence relative to M and Φ_M, if every edge e of G is verified in p.

Proof. In order to prove that $l_i(p)$ is a checking sequence we first assume that $M \overset{l_i(p)}{\equiv} M'$, where $M' \in \Phi_M$. If each edge of G is verified in p then the head of each edge e is t-recognized. This, in turn, implies that there exists subpaths p_{jm} such that the set $\{\text{head}(p_{jm})\}_{m=1,|Z(\varepsilon)|}$ is z-recognized for $j = 1,\ldots,|Q|$.

Now let s_{jm} be those prefixes of $l_i(p)$ which lead to head(p_{jm}). We define a function $f : Q \to Q'$ by letting

$$f(\delta(q_0, s_{j1})) := \delta'(q'_0, s_{j1}) \tag{4}$$

for $j = 1, \ldots, |Q|$. We shall prove below that f is an isomorphism onto a subset of Q'.

By definition of z-recognition, we have $l_i(p_{jm}) = z_m$ and each set $\{\text{head}(p_{jm})\}_{m=1,|Z(\varepsilon)|}$ is distinctly identified as having the common coherency value, say x_j. Therefore, if $j \neq j'$ we must have the group of responses $\{l_o(p_{jm})\}_{m=1,|Z(\varepsilon)|} \neq \{l_o(p_{j'm})\}_{m=1,|Z(\varepsilon)|}$ by definition of the W-set which is a subset of the Z-set. On the contrary, by $l_i(p)$-equivalence λ and λ' are indistinguishable on any segment of $l_i(p)$ which implies that the group of responses defined above also distinguish $|Q|$ distinct states of the automaton M'. The equality $\delta(q_0, s_{jm}) = x_j$ holds for any m by the coherency assumption of z-recognition and hence, in particular, it holds for $m = 1$ which implies that the argument of f in Eq. (4) is the value x_j. As the definition of coherency and the existence of paths p_{jm} ensure that the values on the RHS of (4) are also distinct for different j, the function f is well-defined in $|Q|$ and is an injection. Therefore, in order to prove that f is an isomorphism it is enough to prove that it is a homomorphism.

We define the binary relation R_p on $Q \times Q'$ for the given path p in G by letting $(x, x') \in R_p$ iff for some prefix s of $l_i(p), x = \delta(q_0, s)$ and $x' = \delta'(q'_0, s)$. The following lemma relates R_p to the function f.

Lemma 3. Suppose that $(x, x') \in R_p$ and x is t-recognized in p, then $x' = f(x)$.

Proof of Lemma. Let k be the index for which $x = \delta(q_0, s_{k1})$, which is possible as the latter exhausts all possible states of Q by assumption. Hence, $f(x) = f(\delta(q_0, s_k)) = \delta'(q'_0, s_k)$. As x is t-recognized, there exists subpaths $\{p'_v\}_{v=1,|Z(\varepsilon)|}$ where $\{\text{head}(p'_v)\}_{v=1,|Z(\varepsilon)|}$ is z-recognized and $\{\{x\} \cup \{\text{head}(p'_v)\}_{v=1,|Z(\varepsilon)|}\}$ is a coherent set. Let $\{y'_v\}_{v=1,|Z(\varepsilon)|}$ be the set of states in M' such that $(\text{head}(p'_v), y'_v) \in R_p$. Then by definition of coherency, for each v we have $y'_v = x'$ and therefore $\lambda'(x', z_v) = \lambda'(y'_v, z_v)$. However, as $\{\text{head}(p'_v)\}_{v=1,|Z(\varepsilon)|}$ is z-recognized in p we must have $l_i(p'_v) = z_v$ by definition. We claim that $x' = f(x) = \delta'(q'_0, s_{k1})$ as both yield the same outputs to the Z-set. □

This is because the conditions of Lemma 1 are satisfied by the existence of the subset of paths p_{jm} for which $l_i(p_{jm}) = w_h$ for some $1 \leq h \leq |W|$. Therefore, using Lemma 2 it follows that $\Delta(\varepsilon)$ is atomic, and hence identical responses by two states x' and $\delta'(q'_0, s_{k1})$ of M' to the elements of the Z-set implies the desired equality.

In order to prove that

$$f(\delta(q, \sigma)) = \delta'(f(q), \sigma)$$

$$\lambda(q, \sigma) = \lambda'(f(q), \sigma) \tag{5}$$

for any $q \in Q$ and $\sigma \in \Sigma_i$ we shall use the above lemma and the edge verification properties of p. Clearly the above formulas correspond to an edge e with head(e) $= q$, tail(e) $= \delta(q, \sigma)$ and $l(e) = \sigma/\lambda(q, \sigma)$. As the edge e is verified in p there exists a subpath p' of length one in p such that head(p') is t-recognized as q = head(e) in $p, l(p') = \sigma/\lambda(q, \sigma)$ and tail(p') is t-recognized as tail(e) $= \delta(q,\sigma)$. Let y and y' be uniquely identified as the corresponding states in Q' for head(p') and tail(p'), respectively, i.e. (head(p'),y), (tail(p')y') $\in R_p$.

By Lemma 3 and t-recognition assumptions we have

$$y = f(\text{head}(p'))$$

$$y' = f(\text{tail}(p')). \tag{6}$$

Also as p' is a subpath of p, we have

$$y' = \delta'(y, \sigma) = \delta'(f(q), \sigma) \tag{7}$$

and moreover, as p' verifies the edge e

$$\lambda(q, \sigma) = \lambda(\text{head}(p'), \sigma) = \lambda'(y, \sigma)$$

which proves, after considering $y = f(q)$, the second relation in Eq. (5). Substituting tail(p') $= \delta(q, \sigma)$ and y' given by (7) in the first equation in (6) proves the first relation in Eq. (5) and the proof is complete. □

4.2. Building checking sequences

In this section, we first specify a concrete checking sequence for theoretical purposes and then we discuss the various strategies for lowering its cost. Our formulations will be of the form of finding a minimum length path p in G that starts at q_0 subject to the constraint that a given set of paths $\{p_j\}_{j\in J}$ are subpaths of p. We shall demonstrate that this problem can be transformed into a rural Chinese postman tour (RCPT) problem. We first look at the general case. Let $\{L_{x_j'x_j}\}_{j=1,|Q|}$ be locating sequences ending up at the states $\{x_j\}_{j=1,|Q|}$ of M that cover the entire state set Q of M, and x_j' are arbitrary.

Proposition 6. If a path p in G of a given M that starts at q_0 has subpaths given by

(1) $\{p_{jm}\}_{j=1,|Q|;m=1,|Z(\varepsilon)|}$, where $p_{jm} = p_{1j}.p_{2jm}$ and $\text{head}(p_{1j}) = x_j'$, $l_i(p_{1j}) = L_{x_j'x_j}$ and $l_i(p_{2jm}) = z_m$; and

(2) $\{t_{jkm}\}_{j=1,|Q|;k=1,|\Sigma_i|;m=1,|Z(\varepsilon)|}$, where $t_{jkm} = t_{1j}.t_{2jk}.t_{3jkm}$ and $\text{head}(t_{1j}) = x_j'$, $l_i(t_{1j}) = L_{x_j'x_j}$, $l_i(t_{2jk}) = \sigma_k$ and $l_i(t_{3jkm}) = z_m$,

where the inputs σ_k cover the entire input alphabet Σ_i, then the sequence $l_i(p)$ is a checking sequence.

Proof. We make use of Theorem 1 for the proof. Hence, we have to demonstrate that each edge is verified. Let e be any edge where $\text{head}(e) = x_j$ and $l(e) = \sigma_k/\gamma$. We claim that the path t_{2jk} verifies e. First $\text{head}(t_{2jk})$ is t-recognized by the presence of the subpaths $\{p_{jm}\}_{j=1,|Q|;m=1,|Z(\varepsilon)|}$ by the coherence property of the locating sequence, $L_{x_j'x_j}$, and the z-recognition of the set $\{\text{head}(p_{2jm})\}_{m=1,|Z(\varepsilon)|}$. Similar consideration holds for $\text{tail}(t_{2jk})$ which equals $\text{head}(t_{3jkm})$ for any m, as it is z-recognized—hence t-recognized—by the presence of the subpaths t_{3jkm}. On the contrary, $l(t_{2jk}) = l(e) = \sigma_k/\gamma$ by construction which proves the desired edge verification and hence the theorem. □

The next question one asks is whether further economizing the length of the checking sequence is possible. The answer to this question is affirmative. We first simplify the checking sequence above by replacing the subpaths t_{jkm} by t_{jkm}' where $t_{jkm}' = t_{2jk}.t_{3jkm}$, i.e. t_{jkm} without the initial locating sequence part. Instead we impose the additional condition that $\text{head}(t_{2jk})$ is t-recognized. Here we can make use of Proposition 5 and synthesize the path p by making use of the already verified edges. We summarize this situation by the following proposition, the proof of which is simple and hence omitted.

Proposition 7. If a path p in G of a given M that starts at q_0 has subpaths as in Proposition 6 with the following two modifications:

(1) Each t_{jkm} is replaced by t_{jkm}' as explained above,

(2) The states $\text{head}(t_{2jk})$ are all t-recognized, then $l_i(p)$ is a checking sequence.

How can we further transform this formulation such that it reads as an RCPT problem? Two approaches exist for this transformation. The first one is an extension of the approach suggested in Ref. [13]. The second approach exploits the freedom in choosing a specific tour after solving the RCPT problem—which only fixes the edges used and the number of times they must be traversed for which there are a multiplicity of possible Euler tours—[26]. Before we discuss these approaches we transform the basic graph G into another graph G' in a manner to incorporate the constrained paths as additional edges.

Consider the graph $G = (V,E)$ given by M. Let V' be an exact replica of V called the image set of V. Intuitively, elements of V' shall be used as those elements of V that are t-recognized upon arrival. We now define a new graph $G' = (V \cup V', E \cup E' \cup E'' \cup E''')$. The additional edges are defined below:

(1) A black edge corresponding to an edge $e = (v_i,v_j)$ in E is placed from the image of v_i in V' towards its target v_j in V with the same label as the label of e. Call these additional edges E'. All edges E of G are also black edges. All black edges, whether they start from a t-recognized vertex in V' or not, always end up in an unrecognized vertex in V.

(2) Every path $p_{jm} = p_{1j}.p_{2jm}$ contributes two special and consecutive blue edges: the first edge starts from vertex $\text{head}(p_{1j})$ and ends up in vertex $\text{tail}'(p_{1j})$, and the second edge starts from $\text{tail}'(p_{1j})$ and ends up in $\text{tail}(p_{2jm})$. Here primes refer to the images of vertices in V that are in V'. The fact that an end point vertex is in V' denotes that after a locating sequence, the arrival vertex is t-recognized if all p_{jm} are subpaths of p. We call these set of blue edges E''.

(3) All paths t_{jkm}' are placed as single red edges as follows: from vertex $\text{head}'(t_{2jk})$ in V' to the vertex $\text{tail}(t_{3jkm})$ in V. We call these set of edges E'''.

At this stage our first formulation is to find a minimum-cost path T that starts at q_0 and covers all the edges corresponding to the two consecutive blue edge paths $p_{jm} = p_{1j}.p_{2jm}$ and all the red edges in G'. Here there is no novelty by using t_{jkm}' instead of t_{jkm} as for covering the red edges the first locating sequence edges are essential to reach to V' nodes. We now suggest the two heuristic solutions to the problem of finding a T, defined above making use of Proposition 5. The minimization problem in question is an instance of the RCPT problem which is known to be NP-complete in the general case [23].

Solution 1 (An extension of an idea in Ref. [13]). Choose any spanning tree in G rooted at any desired vertex. Call the edges of this spanning tree *gray* edges and place these edges between the vertices of V' and delete any black edges emanating from a vertex in V' if they coincide in input label with a gray edge emanating from the same vertex. That is, we obtain the graph G'' from G' by adding new gray edges that are represented by the set E'''' and deleting some edges from the set E' as explained above. The

following lemma explains why the problem formulated via G'' can be solved in this manner.

Lemma 4. Let p be any path in G'' that starts at q_0 and covers the subpaths p_{jm} and t'_{jkm}, then both the head and the tail vertices of each gray edge in p is t-recognized.

Proof. We use induction on the depth of the heads and tails of the gray edges relative to the spanning tree. First we observe that all the red edges that verify a gray edge have the same starting vertex in V'. We start with the root vertex of the spanning tree. By construction the only way to arrive at the root vertex is via a locating sequence from a vertex in V and as all p_{jm} paths are covered in p the root is t-recognized by definition. Now assume that all heads and tails of gray edges with depth K are t-recognized. Let j be the head or tail of a gray edge of depth $K+1$ in V'. The only way to arrive at such a vertex is through other vertices of V' via gray edges that emanate from vertices of depth K or smaller. By the induction hypothesis and Preposition 5, the vertex j is t-recognized provided the final gray edge that starts from some vertex, say $i \in V'$, of depth K, that arrives at j is verified. However, as the group of red edges that verify this gray edge also leave from vertex i, its verification is legitimate as each t'_{jkm} is covered in p and the only vertices involved in arriving at vertex i are those of depth K or smaller which are t-recognized by the induction hypothesis. This completes the proof. □

Solution 2 (Solution via selection of an Euler tour [26]). Here we replace the spanning tree of G for the previous solution that defines the gray edges by the entire graph G. Therefore, all edges of G are placed between the vertices in V' with gray colors and all outgoing black edges from V' are removed. After the RCPT problem is solved we try to choose the associated Euler tour to satisfy the t-recognition of nodes prior to application of a transition testing red edge. The tour must be so selected that whenever a gray edge is traversed it is already verified by having traversed all its corresponding red edges. A sufficient condition to guarantee this is given by the following proposition.

Proposition 8. Suppose it is possible to choose the maximal arborescence tree for constructing the Euler tour in such a way that no red edge that is used for verifying a gray edge that occurs in the solution of the RCPT problem belongs to the arborescence tree. Then the input projection of the label of the corresponding path starting from the initial state is a checking sequence.

Proof of Proposition. As an edge in the maximal arborescence tree denotes the last edge to be traversed from the node for which the edge in question is an outgoing one, and as for a given edge of G its red edges used for verification and its corresponding gray edge are incident as outgoing edges of single common node in V' it follows that the tour can be so selected that all the red edges are traversed before the corresponding gray edge is traversed. This implies by Proposition 5 that tails of gray edges are t-recognized relative to this Euler tour and the result follows from Proposition 7. □

If choosing the arborescence tree in the way described in Proposition 8 is not possible we revert to some simple heuristics for adding gray or black edges to the solution of the RCPT problem. As the solution of this problem, in general, already necessitates heuristics, not much is lost by this step and practical examples confirm the efficiency of this pragmatic approach. We refer the reader to the example at the end of this section that illustrates solution 2 described above for the case where a distinguishing sequence exists.

Basically, there is no difference when a problem is transformed into the special case $\varepsilon = 0$. On the contrary, if we further transform it to the special case when a DS exists, slightly simpler and finer solutions are available. We briefly discuss these in the following.

We refer to the single element of W by definition as DS. Theorem 1 induces the path constraints $\{d_j\}_{j=1..|Q|}$ where d_j is the path with head$(d_j) = x_j$ of G; moves according to the input label $l_i(d_j) = \mathrm{DS}.I_j$, where I_j is a minimum length transfer sequence from $\delta(x_j,\mathrm{DS})$ to the final state tail$(d_j) = x'_j$ where x'_j is completely arbitrary. We assume that the problem is solved for all possible choices of x'_j corresponding to each x_j and without loss of generality, henceforth, we fix x'_j at some constant value. The next constraint is the transition verification path segment and we assume that each transition path is followed by the appropriate d_j path for the z-recognition, instead of t-recognition, of the final states. More precisely, the next set of path constraints are of the form p_{jk} where the first index j signifies the start state x_j of the transition to be checked and the second index k codes the input symbol σ_k which fixes the transition. Thus, we have head$(p_{jk}) = x_j$ and $l_i(p_{jk}) = \sigma_k.l_i(d_m)$, where the edge e verified is given as head$(e) = x_j$, tail$(e) = x_m$ and $l(e) = \sigma_k/\lambda(x_j,\sigma_k)$.

Unfortunately, these constraints are not sufficient to solve the problem as the t-recognition of the start state of a transition is not guaranteed. One strategy to cope with this problem is to replace the d_j constraints by $d'_j := d_j.d_{mj}$, where tail$(d_j) = $ head$(d_{mj}) = x_m$. This is made precise by the following proposition.

Proposition 9. Let p be any path in G of a given M that starts at q_0 and assume that for each j, k, d'_j and p_{jk} are subpaths of p. Under these conditions and the equivalence condition $M \overset{l_i(p)}{\equiv} M'$ for any j, k, tail(d_j) and tail(p_{jk}) are t-recognized.

Proof. As tail$(p_{jk}) = $ tail(d_m), it is enough to prove the result for d_j. If d_j occurs in $d'_j = d_j.d_m$, then tail$(d_j) = $ head(d_m) and the latter is z-recognized in d_m and therefore,

t-recognized. If d_j occurs in $d'_v = d_v.d_j$ then we consider the subpath $d'_j = d_j.d_{mj}$ and conclude that $\{\text{tail}(d_j), \text{head}(d_{mj})\}$ is a coherent set as each is preceded by a DS.I_j applied at the same state x_j, and therefore tail(d_j) is t-recognized as head(d_{mj}) is z-recognized. Finally, if d_j occurs in p_{ik} for some i and k then again tail(d_j) = tail(p_{ik}) is t_r recognized using d'_j as in the previous case. □

In spite of modification towards double DS sequence constraints the resulting p still may not qualify to be a checking sequence. In constructing the path p, in addition to the given subpath constraints, we must ensure that every time we enter a path p_{jk}, head(p_{jk}) is t-recognized as this is demanded by the definition of transition verification. However, this is a special case of the problem for which two distinct solutions are suggested above. Note that, unlike the general case, for this special case all blue and red edges used for recognition and verification end up in V' nodes by Proposition 9. In solution 1, the only nodes in V are those entered by black edges. In solution 2, there is no need for the V and V' distinction, hence they can be superimposed on each other. We illustrate the method discussed above via the example given below. In particular, we apply to this example the method proposed in solution 2. Application of solution 1 to this example has been treated elsewhere [13] and will not be repeated here.

Consider the automaton M given in Fig. 1(a). A minimal distinguishing sequence for M is given by DS = aa. Using the procedure described in (1) of Remark 1 we derive the following efficient distinguishing suite: DS(1) = DS(2) = aa and DS(3) = a, where the integer within the parentheses denotes the state at which the reduced distinguishing sequence is applied. In Fig. 1(b), the solution to the RCPT problem is given. Noting the path inclusion relation, *dblue*3 ⊆ *dblue*2, it is enough for the tour to include the extra edges *dblue*1, *dblue*2 for state recognition, where *dbluei* is the single edge that stands for the path resulting in applying *two* consecutive distinguishing sequences at state i which, by Proposition 9, guarantees t-recognition at the node of arrival if the arrival occurs at the end of a blue edge (corresponding to the path for a distinguishing sequence). Similarly, the inclusion relations *redi*1 ⊆ *dbluei* for i = 1,2,3, it is enough to include the paths *redi*2 for i = 1,2,3 for transition verification. Therefore, the rural edges for the RCPT problem are *redi* for i = 1,2,3 and *dbluei* for i = 1,2. Hence, as seen in Fig. 1(b), the only black edge of the original graph G of M used to solve the RCPT problem is the edge from node 1 to 2 with the label a/0, and the edge must be included twice (as shown by the integer in parentheses) so that the degree of each node is neutralized to zero for an Euler tour to exist. It remains to choose the Euler tour for this RCPT solution appropriately so that in the tour, before any black edge is traversed the corresponding red edge must have been traversed for having verified the black edge before it is used. As explained above this can be accomplished if no arborescence [22] edge is chosen as a red edge. This is what we explain next.

In Fig. 2, we show the maximal arborescence for the graph given in Fig. 1(b). Root node is the initial state 1 and by definition of the arborescence tree the last edge, in the tour starting from the root, to leave node e is towards node 1 with a *null* label; the last edge to leave node 3 is towards node e again with *null* label and the last edge to leave node 2 is towards node 3 with the label *dblue*2. Hence, no arborescence edge is red as desired and a feasible Euler tour starting from the root node is: *red*12.*dblue*1.*a*/0.*red*22.*red*32.*a*/0.*dblue*2 *null*.*null* which corresponds to the minimum length path *red*12.*dblue*1.*a*/0.*red*22.*red*32.*a*/0.*dblue*2 with a total cost of 16 units if the cost of each transition is taken as 1 unit.[2,3]

5. The effect of overlapping path segments

In this section, we investigate the problem of exploiting the overlapping property of desired subpaths in synthesizing a path that corresponds to a checking sequence. We say that an ordered pair (p,p') of paths p and p' of G of a given M are overlapping if for $p = e_1 \ldots e_n$ and $p' = e'_1 \ldots e'_{n'}$ there exists an m such that $n - n' + 2 \le m \le n - 1$, and $e_m = e'_1 \ldots e'_{n-m+1}$. In short, a strict postfix of p is a strict prefix of p' (i.e. no path includes the other). In the extreme case the common postfix and prefix may consist of a single edge, namely, the last edge of p is the first edge of p'. A sequence of paths $(p_1,p_2,\ldots,p_n)$ is said to be overlapping if every pair (p_i,p_{i+1}) of paths are overlapping for $i = 1,\ldots,n-1$. The sequence is said to be a *maximal* overlapping one relative to a set of paths P if the sequence cannot be extended in the backward or forward direction by adding another distinct element of P to it. Now let $P = \{p_i\}_{i=1,N}$ be a finite set of paths in G of a given M. We want to solve the problem of computing a minimum cost path p such that for each i, p_i is a subpath of p. We shall transform this into an RCPT problem. Note that the special case of the problem above, where each path consists of a single edge, is already in the standard RCPT form.

We shall assume that no two elements of the set P stand in the relation of being a subpath of the other. If a path in P is a subpath of another path it has null contribution as a constraint, as the latter path already solves the constraint imposed by the former. If such subpaths exist, we simply remove the included paths from the set P. Our strategy of attack for the above problem consists of two steps. In the first step, we atomize parts of the paths above by replacing

[2] If the same approach is applied without using the efficient distinguishing suite based on the DS, the minimal cost of the solution can be shown to be 21 units, hence 25% reduction in cost can be attributed to the use of the efficient distinguishing suite.

[3] It is interesting to note that the solution 1 above, applied to this example yields a checking sequence with a minimum cost of 33 units as reported in Ref. [13].

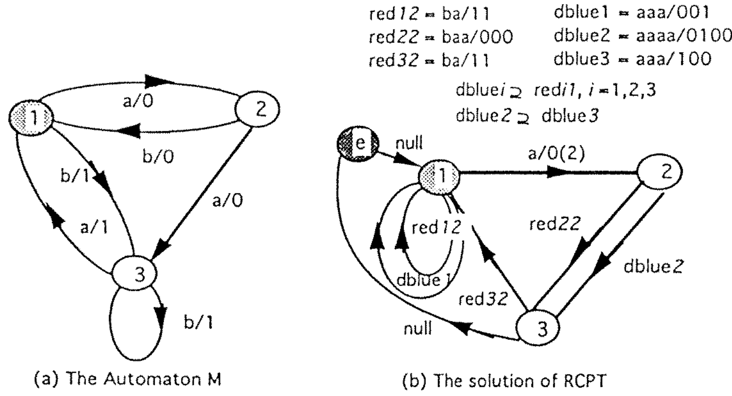


Fig. 1. An automaton M and the solution of the RCPI problem.

them with special externally defined edges in such a way that the atomization—hence, simplification—is the coarsest without destroying potential overlapping opportunities. Clearly the coarsest possible atomization corresponds to the replacement of each path p_i by a special edge whereby the problem reduces to an RCPT problem. Such an atomization may destroy the overlapping possibilities if some of the paths in P overlap in the sense explained above. Hence, in the second step we augment G of M by splitting the vertices in such a way that with the help of the new vertices, covering a path p_i reduces to covering a special edge in the augmented graph, meanwhile economizing the total path length via exploitation of overlapping.

In order to obtain the coarsest atomization we proceed as follows:

For a given maximally overlapping sequence $(p_1,\ldots,p_n)$, where each $p_i \in P$, we decompose the union Y_k of these paths into a minimum number of concatenated subpaths, i.e.

$$Y_k = p_{1k}.p_{2k}.\ldots.p_{n_k k}$$

such that for each p_i and each p_{jk}, p_{jk} is either subpath of p_i or $p_{jk} \cap p_i = \emptyset$. Repeating this for each maximally overlapping sequence of P, the coarsest refinement is obtained in the following way[4]:

1. if a path p_i is common to more than one group, say Y_k and $Y_{k'}$, then we make the necessary refinement by path splitting such that every subpath p_{ik} of this common path that appears in one of the concatenations above also appears as a subpath $p_{i'k'} = p_{ik}$ in the other concatenation and vice-versa;
2. we first assign concatenated transition edges e_{ik} to concatenated subpaths p_{ik}, such that $\text{head}(e_{1k}) = \text{head}(p_{1k}), \text{tail}(e_{n_k k}) = \text{tail}(p_{n_k k})$, all other intermediate vertices of the concatenated edges e_{ik} are newly defined and the label of each e_{ik} is the entire label of the path p_{ik} and its cost defined as the entire cost of the path given by p_{ik}. Next we assign for each path $p_j = p_{i_j k},\ldots,p_{i'_j k}$ for which $i_j \neq 1$ an edge e'_{jk} such that $\text{head}(e'_{jk}) = \text{head}(p_{i_j k}), \text{tail}(e'_{jk}) = \text{head}(e_{i'_j k})$ and the label of e'_{jk} is equal to $l(e_{i_j k},\ldots,e_{i'_j - 1k})$, and similarly for cost.

The augmented graph G' is obtained by using these augmented edges $\{e_{ik},e_{jk}\}$ in the way described above. The head of every special edge e_{ik} added above corresponds to a state given by $x = \text{head}(p_{ik})$ in the original graph G. For every transition edge from x to y with label l in the original graph we add at the intermediate vertices of the path $e_{1k},\ldots,e_{n_k k}$ (i.e. for $i \neq 1$ and $i \neq n_k$), an edge from $\text{head}(e_{ik})$ to state y with a label l. Finally, a path may appear in more than one distinct maximal overlapping set. If this is the case then:

We superimpose all the edges e_{ik} for different values of k that correspond to subpaths of this common path,

Similarly, we superimpose the edges $e'_{jk} and e'_{j'k'}$ if they correspond to the same shared path.

In the augmented graph every path constraint contributes one and only one edge constraint in the RCPT problem, namely the edge $e_{i'_j k}$ corresponding to the path $p_j = p_{i_j k}\ldots p_{i'_j k}$. This completes the entire description of the augmented graph $G' = (V',E')$.

We illustrate the above description in Fig. 3, where (a) corresponds to a case with three overlapping paths, and (b) corresponds to the same situation with cyclic termination. In part (a) of the figure, the augmented new edges on the overlapping path are shown with $e1, e'1, e2, e'2, e3$ where the beginning node $h1$ and the end node $t3$ belong to the graph G, whereas all the other intermediate nodes are new. We denote the intermediate nodes by $h'2, t'1, t'2 and h'3$ to remind that they are mirror images of the corresponding non-primed nodes of G. The edges to be

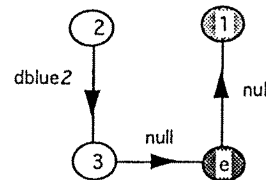


Fig. 2. Maximal arborescence.

[4] In case the sequence is cyclic, slight modifications are required in the procedure explained. The cyclic case is illustrated in Fig. 3(b) below.

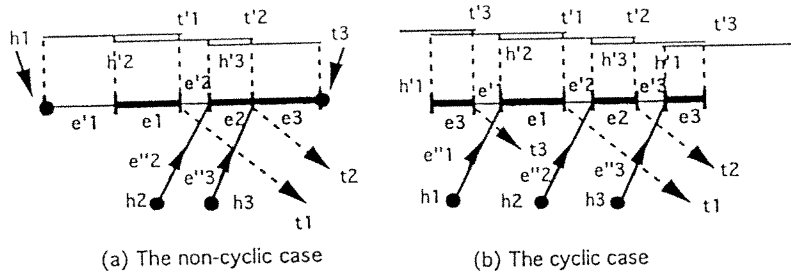


Fig. 3. Overlapping augmentation.

included in the RCPT are $e1$, $e2$ and $e3$ representing the three overlapping paths that they terminate. The paths that are used in climbing into the overlapping zone are represented by the edges $e''2$ and $e''3$ in the figure. The label of $e''2$ is equal to the concatenation of the labels of $e1$ and $e'2$ and similar labelling holds for $e''3$. Finally, exit from the intermediate region of the overlapping zone is possible via null label and null cost edges from $t'1$ to $t1$ and $t'2$ to $t2$, respectively, as shown in the figure. In part (b) of the figure, we illustrate the case where the last path cyclically overlaps with the first path. The overlapping zone is a cycle of edges, starts at node $h'1$ and completes the loop with the final edge $e'3$, and the rest as simple modification of the previous case.

Finally, we extend the example given Fig. 1 of the previous section by exploiting overlapping. We first observe that the path sequences (*dblue*1,*dblue*2) and (*dblue*2,*dblue*1) are both maximal cyclic overlapping sequences and no others exist. These cyclic overlappings are captured by the following augmentation given in Fig. 4. The edges to be included in the RCPT formulation are $e2$ and $e3$. The labels of the climbing edges that relate the loop to the actual graph are indicated in the figure. The loop replaces the *dbluei* edges of the previous solution. Red edges remain intact, as they do not overlap. The solution after augmenting the original graph with the loop given in Fig. 4 is given in Fig. 5. We note that the red edge that occurs in the arborescence tree is not used in verifying the only gray edge in the example, which is between nodes 1 and 2 and denoted by its label $a/0$. Therefore, by Propositions 8 and 9, the conditions for legitimate edge verification are maintained. The optimal path is given by $red12.a/0.e'.e2.e3.a/0.red22.red32$ with a total cost of 14 units.

6. Conclusions

We have presented a general model for constructing minimal length checking sequences from a given deterministic FSM M. The model is based on characteristics of checking sequences and sets of state identification and verification sequences. The generality of the model lies in the fact that it relaxes the common assumptions made in the literature on M and its implementation M', and that it utilizes a distinguishing suite which is composed of either distinguishing sequences, characterization sets, or unique input–output sequences depending on their existence. Moreover, the model is general in the sense that it does not depend on an a priori choice of subpaths that collectively identify the states of M'. Further, the connectivity of the underlying digraph representing the model is guaranteed to obtain a rural Chinese postman path. Therefore, the proposed model provides a means of obtaining a globally optimal solution to the construction of a checking sequence of minimal length.

The proposed model and related heuristics lend themselves to overlapping segments of input sequences in an attempt to construct checking sequences of optimal length. To the best of our knowledge, this is the first method that uses the concept of overlapping introduced by Hennie [3] for the construction of checking sequences (not "test sequences" as studied earlier in the literature). Our experiments show that wherever possible, the use of overlapping reduces the length of checking sequences 10–15% of the length of the sequences obtained without overlapping.

Appendix A

In order to prove Proposition 4 we start with the following lemma, the proof of which is straightforward and omitted.

Lemma 5. Let $B = (X,\Sigma_i,\Sigma_o,\delta'',\lambda'',x_0)$ be any Mealy automaton with m states and let t, $s \in \Sigma_i^*$. Then for any $x \in X$ there exists an integer k such that $0 \le k < m$ and $\delta''(x,t.s^k) = \delta''(x,t.s^m)$.

The next lemma makes use of the previous one to fix the system given by automaton B to a single state prior to the application of each a_i, in the absence of a reset feature.

Lemma 6. Let $\{a_i\}_{i=1,N}$ be a set of input sequences and consider the forward recursion given by

$$\theta_{i+1} = \theta_i.(a_i.\theta_i)^m,\ i = 1,\dots,N-1,$$

where $\theta_1 := e = emptytrace$. Under these conditions for any $x \in X$ there exists an integer $k(i)$, such that if r_i is the solution of the backward recursion

$$r_i = r_{i+1}.f_i(\theta_i),\ i = N-1,\dots,1$$

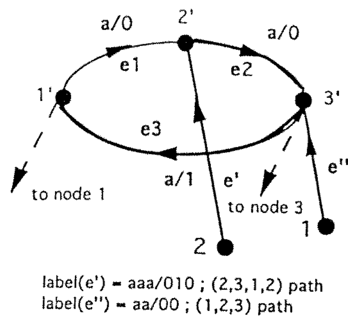


Fig. 4. Cyclic overlapping.

where $r_N = e$ and f_i is given by

$$f_i(\theta_i) := \begin{cases} \theta_i.(a_i.\theta_i)^{k(i)}.a_i & \text{if } k(i) \geq 0 \\ e = emptysequence & otherwise \end{cases}$$

such that the following properties are enjoyed by the sequences of traces θ_i, r_i:

1. $\delta''(x, r_i.\theta_i) = \delta''(x, \theta_N)$, $i = 1,\ldots,N-1$,
2. $r_i.\theta_i.a_i$ is a prefix of $r_{i+1}.\theta_{i+1}$ for $i = 1,\ldots,N-1$.

Proof. Using the forward recursion expression for θ_{i+1}, we obtain:

$$r_{i+1}.\theta_{i+1} = r_{i+1}.\theta_i.(a_i.\theta_i)^m.$$

Assuming $x \in X$ is given we apply Lemma 5 with $t := r_{i+1}.\theta_i$ and $s := (a_i.\theta_i)$ and depict the existence of an integer k such that $0 \leq k < m-1$ and

$$\delta''(x, r_{i+1}.\theta_i.(a_i.\theta_i)^m) = \delta''(x, r_{i+1}.\theta_i.(a_i.\theta_i)^k).$$

Now set $k(i) := k-1$, then if $k \neq 0$ then $k(i) \geq 0$ and by definition of f_i

$$\delta''(x, r_{i+1}.\theta_i.(a_i.\theta_i)^k) = \delta''(x, r_{i+1}.\theta_i.(a_i.\theta_i)^{k(i)}.a_i.\theta_i)$$

$$= \delta''(x, r_i.\theta_i)$$

where we substituted for r_{i+1} using the backward recursion. If, on the contrary, $k = 0$, then $k(i) = -1$ and $f_i = e$ by definition, which again yields the same result. This proves (1) of the lemma. To prove (2) simply note that

$$r_{i+1}.\theta_{i+1} > r_{i+1}.\theta_i.(a_i.\theta_i)^m > r_{i+1}.\theta_i.(a_i.\theta_i)^{k(i)}.a_i.\theta_i.a_i$$

$$> r_i.\theta_i.a_i$$

if $k \neq 0$. If $k = 0$, $r_{i+1} = r_i$ and the result again holds. □

The complexity of the input sequence θ_N in Lemma 6 is given by the next lemma.

Lemma 7. The length of the sequence θ_N is given by

$$|\theta_N| = m \sum_{j=1}^{N-1} (m+1)^{(N-1-j)}.|a_j| \qquad (8)$$

In particular, if $l := |a_j|$, for all $j = 1,\ldots,N$, then

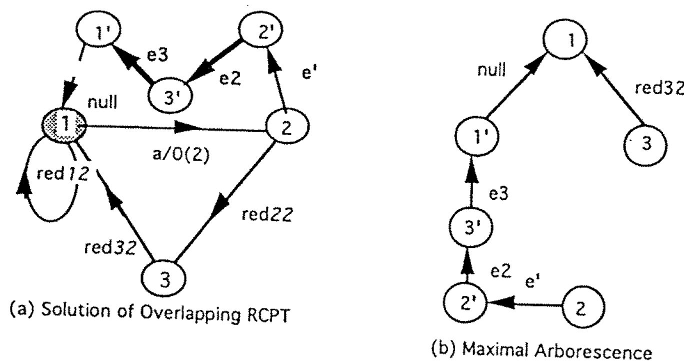
$$|\theta_N| = l.(1 + (1+m)^{N-1}) \qquad (9)$$

The proof involves straightforward computation and is omitted.

Now assume that the automaton M has $n := |Q|$ states and the black box automaton M' has states not larger than $n + \varepsilon$. Recalling the definition of Z, let $z_1,\ldots,z_{|Z(\varepsilon)|}$ denote the elements of Z. For any state q of M define $a_i := z_i.I_i$ for $i = 1,\ldots,|Z(\varepsilon)|$, where I_i is a transfer sequence for which $\delta(q, z_i.I_i) = q$. This choice of a_i has the following consequence.

Lemma 8. If $B = M$ in Lemma 6 (in particular $\delta'' := \delta$, $m := n$ and $N := |Z(\varepsilon)|$) and the values of a_i are as above, then $\delta(q, r_i.\theta_i) = q$ for all $i = 1,\ldots,|Z(\varepsilon)|$.

Proof. We claim that each θ_i and r_i are of the general form $h_1^{i_1}.h_2^{i_2}\ldots.h_K^{i_K}$, where each $h_i \in \{a_i,\ldots,a_N\}$. This follows from the forward recursive definition of θ_i and substituting this form for θ_i in the backward recursion for r_i similar result is also proved for r_i. However, as $\delta(q,a_i) = q$ by the construction of each a_i above the desired result follows as $r_i.\theta_i$ is of the same form as θ_i and r_i. □



(a) Solution of Overlapping RCPT

(b) Maximal Arborescence

Fig. 5. Solution with overlapping.

Proof of Proposition 4. We define the *locating sequence* as $L_{qq'} := \theta_{|Z(\varepsilon)|}.a_{|Z(\varepsilon)|}$, where θ_i is defined as in Lemma 6 with the a_i as above, $N := |Z(\varepsilon)|$, and $m := n + \varepsilon$. If this sequence is applied at any arbitrary state x of the black box, which has at most $n + \varepsilon$ states, then by applying Lemma 6 to the black box, i.e. with $B = M'$, r_i can be selected in such a manner to satisfy (1) of the lemma, i.e.

$$\delta'(x, r_i.\theta_i) = \delta'(x, \theta_{|Z(\varepsilon)|}), i = 1, \ldots, |Z(\varepsilon)| - 1,$$

and as demonstrated in the proof of Lemma 6, $r_i\theta_i$ is a prefix nested sequence with the largest element $\theta_{|Z(\varepsilon)|}$. Moreover, (2) of the lemma shows that the structure of this common state value, say x^*, is such that each z_i is applied at this state.

Now assume that p is given and $L_{qq'}$ occurs a number of times in $l_i(p)$, where the path p satisfies the conditions of Lemma 1 and M is $l_i(p)$-equivalent to M'. By Lemma 8 and part (2) of Lemma 6, in the M automaton the response set to the elements of Z in every occasion of application of $L_{qq'}$ within $l_i(p)$ are identical and equal to the response given at state q. This implies that the response set to the elements of Z of every fixed state of M' corresponding to an occasion of $L_{qq'}$ are also identical. If, on the contrary, the two states x, y of Q' yield the same response set to all the elements of Z then they must belong to some partition $\Delta(\varepsilon)$. However, as the assumption of Lemma 1 holds by the choice of p, it follows from Lemma 2 that $\Delta(\varepsilon)$ must be atomic. Therefore, it must be the case that $x = y$. This proves that the states reached at the end of each $L_{qq'}$ sequence are the same and the coherency conclusion follows. □

Finally, if $L_{vv'}$ is derived from $L_{qq'}$ by attaching a prefix and a postfix to it, then the prefix is irrelevant as long as the path condition for $L_{qq'}$ is satisfied and the fact that the final state sets are coherent follows from Proposition 1, part (c).

Note that the size of the locating sequence $L_{qq'}$ is given by the result of Lemma 7. For the simplest special case where $\varepsilon = 0$ and $|W| = 2$ the length is $l.(2 + (1 + |Q|)$ if $|w_1.I_1| = |w_2.I_2| = l$.

References

[1] A. Gill, Introduction to the Theory of Finite-State Machines, McGraw Hill, New York, 1962.

[2] G. GönenÇ, A method for the design of fault detection experiments, IEEE Transactions on Computers 19 (1970) 551–558.

[3] F.C. Hennie, Fault detecting experiments for sequential circuits, in: Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design, 1964, Princeton, NJ, pp. 95–110, 1964.

[4] M.P. Vasilevskii, Failure diagnosis of automata, Kibernetika 4 (1973) 98–108.

[5] Z. Kohavi, Switching and Finite Automata Theory, McGraw Hill, New York, 1978.

[6] W.Y.L. Chan, S.T. Vuong, M.R. Ito, An improved protocol test generation procedure based on UIOs, Proceedings of ACM SIGCOMM 89 (1989) 283–294.

[7] T.S. Chow, Testing software design modeled by finite state machines, IEEE Transactions on Software Engineering SE-4 (1978) 179–187.

[8] A.T. Dahbura, K.K. Sabnani, M.Ü. Uyar, Formal methods for generating protocol conformance test sequences, Proceedings of the IEEE 78 (1990) 1317–1325.

[9] S. Fujiwara, et al., Test selection based on finite state models, IEEE Transactions on Software Engineering SE-17 (6) (1991) 591–603.

[10] A. Rezaki, H. Ural, Construction of checking sequences based on characterization sets, Computer Communications 18 (12) (1995) 911–920.

[11] K.K. Sabnani, A.T. Dahbura, A protocol test generation procedure, Computer Networks and ISDN Systems 15 (4) (1988) 285–297.

[12] D.P. Sidhu, T.K. Leung, Formal methods for protocol testing: a detailed study, IEEE Transactions on Software Engineering SE-15 (1989) 413–426.

[13] H. Ural, X. Wu, F. Zhang, On minimizing the lengths of checking sequences, IEEE Transactions on Computers 46 (1) (1997) 93–99.

[14] M. Yannakakis, D. Lee, Testing finite state machines: fault detection, Journal of Computer and System Sciences 50 (1995) 209–227.

[15] A.V. Aho, et al., An optimization technique for protocol conformance test sequence generation based on UIO sequences and rural Chinese postman tours, IEEE Transactions on Communication COM-39 (11) (1991) 1604–1615.

[16] S.C. Boyd, H. Ural, On the complexity of generating optimal test sequences, IEEE Transactions on Software Engineering SE-17 (1991) 976–978.

[17] M.S. Chen, Y. Choi, A. Kershenbaum, Approaches utilizing segment overlap to minimize test sequences, in: Proc. PSTV '90, Ottawa, Canada, 1990, pp. 67–84.

[18] R.E. Miller, S. Paul, On the generation of minimum-length conformance tests for communication protocols, IEEE/ACM Transactions on Networking 1 (1) (1993) 116–129.

[19] Y.N. Shen, F. Lombardi, A.T. Dahbura, Protocol conformance testing using multiple UIO sequences, IEEE Transactions on Communications COM-40 (1992) 1282–1287.

[20] H. Ural, K. Zhu, Optimal length test sequence generation using distinguishing sequences, IEEE/ACM Transactions on Networking 1 (3) (1993) 358–371.

[21] B. Yang, H. Ural, Protocol conformance test generation using multiple UIO sequences with overlapping, in: Proc. ACM SIGCOMM '90, Philadelphia, USA, 1990, pp. 118–125.

[22] J.A. Bondy, U.S.R. Murty, Graph Theory With Applications, Elsevier, New York, 1976.

[23] J. Edmonds, E.L. Johnson, Matching euler tours and the Chinese postman, Mathematical Programming 5 (1973) 88–124.

[24] M.K. Kuan, Graphic programming using odd or even points, Chinese Mathematics 1 (1962) 273–277.

[25] D. Lee, M. Yannakakis, Testing finite state machines: state identification and verification, IEEE Transactions on Computers 43 (1994) 306–320.

[26] H. Çalgm, Implementation of an optimal conformance test sequence generation using distinguishing sequences and modified rural postman tour algorithm, Master's thesis, Middle East Technical University, Ankara, Turkey, 1996.

Information and Software Technology Notes for Authors

(also available at: www.elsevier.nl/locate/infsof)

Please follow these instructions carefully to ensure that the review and publication of your paper is as swift and efficient as possible. These notes may be copied freely.

Aims and scope

Information and Software Technology is the international technical journal covering software development. The Journal addresses the problems that arise when developing, maintaining, and modifying information systems in today's organizations. It provides up-to-date information on the latest software development research.

Submissions

Contributions falling into the following categories will be considered for publication.

- Original high-quality research and review papers (preferably no more than 20 double line spaced manuscript pages, including tables and illustrations)
- Short communications, for rapid publication (no more than 10 double line spaced manuscript pages including tables and figures).

Information and Software Technology also publishes book reviews and conference reports. Contributions for the calendar of forthcoming events should be sent to the Group Editor.

Contributions are normally received with the understanding that they comprise original, unpublished material and are not being submitted for publication elsewhere. Translated material, which has not been published in English, will also be considered. All submissions should be accompanied by a written declaration, signed by all authors, that the paper has not been submitted for consideration elsewhere. Authors are solely responsible for the factual accuracy of their papers. The receipt of manuscripts will be acknowledged.

Address for submissions – Papers may be submitted to either of the Co-editors
Professor Martin Shepperd, Department of Computing, Bournemouth University, Poole House, Talbot Campus, Fern Barrow, Poole, Dorset BH12 5BB, UK.
Michael Dyer, Dycon Systems, Suite 780, 7201 Wisconsin Avenue, Bethesda, MD 20814, USA.

Any queries regarding accepted papers, proofs or offprints should be addressed to Elsevier Science Ireland Ltd., Elsevier House, Brookvale Plaza, East Park, Shannon, Co. Clare, Ireland. Tel.: (+353) 61 709600; Fax: (+353) 61 709114.

Review process

All contributions are read by two or more referees to ensure both accuracy and relevance, and revisions to the script may thus be required. On acceptance, contributions are subject to editorial amendment to suit house style. When a manuscript is returned for revision prior to final acceptance, the revised version must be submitted as soon as possible after the author's receipt of the referees' reports. Revised manuscripts returned after four months will be considered as new submissions subject to full re-review.

Copyright

The submission of a paper will imply that, if accepted for publication, it will not be published elsewhere in the same form, in any language, without the consent of the Publisher. Before publication, authors are requested to assign Copyright to Elsevier Science B.V. to sanction reprints and photocopies, and to authorize the reprinting of complete issues or volumes according to demand. It is the author's responsibility to obtain written permission to quote material that has appeared in another publication.

Disk submission

Information and Software Technology welcomes contributions prepared on disk. Disks should be sent with the final revised manuscript, and not the original submission. Please contact the editorial office for full instructions or refer to back issues of the journal for notes on disk preparation.

Preparation of scripts

You should write in clear and concise English. Spelling should follow the Oxford English Dictionary. Authors whose native tongue is not English are assured that in-house editorial attention to their contributions will improve clarity and acceptability to readers. Four copies of each article are required, typed on one side of A4 only, in double spacing (including abstract and references) and with wide margins. Please number every sheet of paper. Authors are responsible for ensuring that all manuscripts (whether original or revised) are accurately typed before final submission. Manuscripts will be returned to the authors with a set of instructions if they are not presented according to these Notes for Authors.

Arrangement of papers

You should arrange your contribution in the following order:

1. Paper title, author's name, affiliation, full postal address and telephone and fax numbers. Affiliations and addresses of co-authors should be clearly indicated. The title should be short, specific and informative.
2. Self-contained abstract of approximately 100 words, outlining in a single paragraph the aims, scope and conclusions of the paper; three keywords, for indexing purposes.
3. The text, suitably divided under numbered headings.
4. Acknowledgements (if any).
5. References (double spaced, and following the journal style).
6. Appendix (if any).
7. Tables (each on a separate sheet).
8. Captions to illustrations (grouped on a separate sheet or sheets).
9. Illustrations, each on a separate sheet containing no text, and clearly labelled with the journal title, author's name and illustration number.

Style of text

Subdivide your paper in the simplest way possible, consistent with clarity. The text should usually follow the standard sequence of introduction, method, results and discussion. Headings and subheadings for different sections of the paper should be clearly indicated and numbered appropriately. Ensure that all figures and tables are mentioned in the text, and that all references are cited in number order. Note that trade names should have an initial capital letter.

Units and abbreviations

All measurements and data should be given in SI units, or if SI units do not exist, in an internationally accepted unit. If you use any symbol or unit that may not be generally recognized, please include an explanatory footnote the first time it is used, to help the referees, editors and readers. It is also helpful to identify Greek symbols by name in the margin the first time they appear. Abbreviations and acronyms should only be used for unwieldy terms and names which occur frequently in the manuscript. Abbreviations should be used consistently throughout the text, and must be clearly defined in full on first use.

Mathematical and technical setting

Detailed mathematical discussion should be placed in an appendix. Equations and formulae should be typewritten wherever possible. Equations should be numbered consecutively with Arabic numerals in parentheses on the right hand side of the page. Special symbols should be identified in the margin, and the meaning of all symbols should be explained in the text where they first occur. If you use several symbols, a list of definitions (not necessarily for publication) will help the editor. Type or mark mathematical equations exactly as they should appear in print. Journal style for letter symbols is as follows: variables, *italic type* (indicated by underlining); constants, roman type; matrices and vectors, **bold type** (indicated by wavy underlining).

Tables

Tables should be numbered consecutively in Arabic numerals, and given a suitable caption. All table columns should have an explanatory heading, and, where appropriate, units of measurement. Footnotes to tables should be typed below the table, and should be referred to by superscript letters. Avoid the use of vertical rules. Tables should not duplicate results presented elsewhere in the manuscript, e.g. in graphs.

Illustrations

All graphs, photographs, diagrams and other drawings (including chemical structures) should be referred to as Figures, and numbered consecutively in Arabic numerals. All illustrations must be clearly labelled with the journal title, author's name and figure number.
Illustrations should be provided in camera ready form, suitable for reproduction without retouching. They will be photographically reduced in size, typically to fit one or two columns of the journal (approximately 85 or 175 mm), and this should be borne in mind to ensure that lines and lettering remain clear and do not break up on reduction. After reduction, lettering should be approximately the same size as the type used for the text in the journal.
Please ensure that all illustrations within a paper are consistent in style and quality.
A table is usually more effective than a graph or a paragraph of text for recording data.

Graphs and line drawings

The minimum amount of descriptive text should be used on graphs and drawings; label curves, etc. with single-letter symbols (i.e. a, b, c, etc.) and place descriptive matter in the figure caption. Scale grids should not be used in graphs unless required for actual measurements. Please use a selection of the following symbols on graphs: +, ×, □, ○, ■, ●, ▲, ▼. Graph axes should be labelled with the variable written out in full, along the length of the axis, with the unit in parentheses (for example, Length of sample (mm)). Lower case letters should be used throughout, with an initial capital letter for the first word only.
If your illustrations are computer generated, please supply the blackest possible laser output.

Photographs

Supply four sets of black and white prints. If necessary, a scale should be marked on the photograph. Please note that photocopies of photographs are not acceptable. Colour reproduction is available if the author is willing to bear the additional incremental reproduction and printing costs. Please contact the editorial office for details. A letter confirming the author's willingness to accept these costs should be sent with the revised manuscripts.
Authors should note that illustrations will not be returned unless specifically requested.

References

In the text, indicate references to the literature by Arabic numerals in square brackets which run consecutively through the paper. If you cite a reference more than once in the text, use the same number each time. References should follow the style in the journal. Please ensure that references are complete, i.e. that they include, where relevant, author's name, journal or book title, year of publication, volume number, page number, editors, publishers and place of publication. If in doubt, please include all available information. Take care that references are mentioned in the text in the correct number order.

Proofs

Correspondence and proofs for correction will be sent to the first named author unless otherwise indicated.
Proofs should be checked carefully for typesetting errors, and any queries should be answered in full. It is important that proofs are returned within the time stated or publication will be delayed/errors may not be corrected. Authors may be charged for any alterations other than typesetters' errors.

Offprints

The corresponding author will receive 50 offprints of the paper and one copy of the journal free of charge. Extra copies of offprints, minimum 50, can be ordered when proofs are returned.

EK 12

ÖZ

SDL VE ERPAL DİLLERİNİ KULLANARAK YAPILAN SÜREÇ BELİRTMELERİNİN KARŞILAŞTIRILMASI

Ali SEZGİN
Yüksek Lisans Tezi, Elektrik ve Elektronik Mühendisliği Anabilim Dalı
Tez Yöneticisi: Prof. Dr. Kemal İNAN
Eylül, 1994, 145 sayfa.

Veri haberleşmesindeki artan karmaşıklık ile biçimsel betimleme tekniklerine olan ihtiyaç her zamankinden daha önemli bir duruma geldi. Bu yüzden, günümüzün en önemli araştırma konularından biri, etkili bir biçimsel betimleme tekniği bulmaktır.

Bu tezde, yeni bir biçimsel betimleme tekniği dili olan ERPAL'in spesifikasyon gücü ve esnekliği, bazi spesifikasyonları ERPAL'de yazmak suretiyle gösterilmiştir.

ERPAL'e özel olan, olay denetlemenin kullanımını göstermek için Milner'ın zamanlama problemi anlatıldı. Protokol spesifikasyonunun inceliklerini göstermek için, en basit protokollerden biri olan zebani oyunu örnek olarak verilmiştir. Daha karmaşık bir örnek olarak, OSI'nin basit bir sürümü olan Inres düsünülmüştür. Son olarak, LOTOS, CSP, Estelle, SDL gibi çok kullanılan dillerin onünde hala büyük bir engel olan hareketli ağların spesifikasyonu, ERPAL'ın bu işte nasıl kullanıldığını göstermek için verilmiştir.

Cebirsel bir dil olan ERPAL ile genişletilmiş sonlu durum makinasını temsil eden SDL arasındaki farkları ve benzerlikleri göstermek için, bazı spesifikasyonlar aynı zamanda SDL ile de yapılmıştır.
Anahtar sözcükler: Biçimsel Tanımlama Teknikleri, Cebirsel Diller, SDL, ERPAL

Bilim Dalı Sayısal Kodu: 619.02.02

ÖZ

YENİ BİR CEBİRSEL DİL ERPAL VE SDL-92'NİN BİR ALTKÜMESİNİN ERPAL'E ÇEVİRİLMESİ

Hüsnü YENİGÜN
Yüksek Lisans Tezi, Elektrik ve Elektronik Mühendisliği Anabilim Dalı
Tez Yöneticisi: Assoc. Prof. Semih BİLGEN
Ocak, 1995, 158 sayfa.

Bu tezde, Sonlu Kendi kendini çağıran Süreç cebirinin ([1, 2, 3]) işlemcilerine dayalı olan, yeni bir cebirsel süreç belirtim dili ERPAL (Geliştirilmiş Kendi kendini çağıran Cebirsel Süreç Dili) yazım kuralları ve işlemcilerinin işlevsel anlamları verilerekten tanımlanmıştır. Ayrıca SDL yapılarından ERPAL yapılarına çeviri için gerekli olan kurallar da verilmiş ve açıklanmıştır. ERPAL eşzamanlama özellikleri açısından bakıldığında diğer yaygın olarak bilinen CSP ([4]) ve LOTOS ([5, 6]) gibi dillere nazaran ifade gücünün daha yüksek olduğu görülmektedir. ERPAL şu anda SDL92-to-C++ çeviricisinin tasarım aşamasında fikirsel düzeyde bir ara veri (süreç) yapısı olarak kullanılmaktadır.

Anahtar sözcükler: Ayrık Olay Sistemleri, Formal Tanımlama Teknikleri, Cebirsel Diller, SDL.

Bilim Dal Saysal Kodu: 619.02.02

ÖZ

BİR SONLU DURUM MAKİNASI İÇİN TANIMLAYICI KÜMELER KULLANARAK TEST DİZİSİ ÜRETİMİ

Gücün, Onur Mehmet

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Kemal İNAN

Ocak 1996, 70 sayfa

Bu tezde, bir model sonlu durum makinası için ayırt edici dizi bulunamadığı genel durumda, eniyi test dizisi üretimi problemi incelenmiş ve gerçekleştirilmiştir. Çalışma aynı zamanda, muhtemelen hatalı bir uygulamanın belirtimine göre daha fazla durum sayısına sahip olabileceği koşulda, bir sonlu durum makinası için test dizisi üretiminin araştırmasını da kapsamaktadır.

Anahtar Kelimeler: Sonlu Durum Makinası, Test, Test Dizisi, Kontrol Dizisi, Ayırt Edici Dizi, Tanımlayıcı Küme, Ayırt Edici Küme, Kırsal Postacı Turu

ÖZ

SDL'92 DERLEYİCİSİ ÖN-YÜZÜNÜN TASARIMI VE GELİŞTİRİLMESİ

Doyuran, Ahmet Uygar

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Kemal İnan

Ortak Tez Yöneticisi: Dr. Vladimir Levin

Mayıs 1996, 144 sayfa

SDL belirtimlerinin doğrulanması ve benzetimi için, bu belirtimlerin çeşitli dillere çevrilmesi gerekir. Bu amaçla, SDL'92 derleyicileri için ortak bir ön-yüz geliştirilmiştir. Ön-yüz bir SDL belirtimini alarak buna eşdeğer ağaç yapısına dayanan bir ara gösterim oluşturur. Bu ara gösterim hedef dilden bağımsız olup hedef dil üretimine uygun bir yapıdadır. Bu çalışmada ön-yüzün tasarım ve geliştirilmesi ile ilgili detaylar verilmiştir. Ön-yüzün her aşaması açıklanmıştır, ara gösterimin yapısı anlatılmıştır. Oluşturulan modüllerin test yöntemi açıklanmıştır.

Anahtar Kelimeler: derleyici, ön-yüz, biçimsel belirtim, yazılım mühendisliği, iletişim yazılımı, ayrıştırıcı, SDL

ÖZ

ISDN 2. KATMANININ SDL 92 KULLANILARAK BELİRTİMİ VE GEÇERLİLİK SINAMASI

Çenberci, Doğuş
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Kemal İnan

Ocak 1997, 115 sayfa

Bu tez çalışmasında iki iş başarılmıştır: İlk olarak, Tümleşik Hizmetler Sayısal Şebekesi (ISDN) kullanıcı-şebeke arabirimi veri bağlantı katmanı (Katman 2) hizmetlerinin, SDL aracı Object Geode kullanılarak belirtimi gerçekleştirilmiştir. Daha sonra bu belirtim, geçerliliğinin sınanmasının yanısıra, SDL aracının benzetim ve çözümleme yeteneklerinin keşfedilmesi amacı ile, çeşitli test senaryoları için mesaj sırası grafikleri (MSC) ile birlikte çözümlenmiştir.

Anahtar Sözcükler : ISDN Katman 2, SDL, MSC, Object Geode

ÖZ

XPRESS TAŞIYICI PROTOKOLÜNÜN SDL 92 KULLANILARAK BELİRTİMİ VE GEÇERLİLİK SINAMASI

DALKIRAN, İsmail

Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. KEMAL İnan

Nisan 1997, 144 sayfa

Bu tezde, SDL aracı SDT 3.02 ile Xpress Taşıyıcı Protokolü (XTP) belirtimlenmiştir. Bu belirtim, SDT'nin Geçerleyici ve Benzetici araçlarının sınama kabiliyetleri sayesinde mesaj sırası grafikleriyle (MSC) tanımlanmış test senaryoları ve durum uzayı keşif teknikleri kullanılarak analiz edilmiştir.

Anahtar Sözcükler : Express Taşıyıcı Protokolü, SDL, MSC, SDT

ÖZ

SDL SİSTEMLERİNİN KISMİ SIRALAMA METODLARI İLE DOĞRULANMASI

Şen, Mehmet Alper
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Kemal İnan

Mayıs 1997, 99 sayfa

Kısmi sıralama metodları endüstriyel boyuttaki sistemlerin biçimsel olarak doğrulanmasında kullanılan durum alanı ufaltım tekniklerinden biridir. Kısmi sıralama metodlarından yararlanan sürekli kümeler metodunun SDL programlarına uygulanması ile SDL sistemleri için karışıklıktan kurtaran teknikler türetilmiştir. Bu nedenle SDL programlarını, SDL dilinin detaylarını soyutlayarak, modelleyen SSM adlı bir otomat tanımlanmıştır. Haberleşen SSM'ler için sürekli kümelerin gerekli ve yeterli karakterizasyonu türetilmiştir ve bu da sürekli küme hesaplanmasında bir algoritma formüle edilmesinde kullanılmıştır. Sonuçların türetildiği SDL alt kümesi *save* ve *priority inputs* gibi giriş dizisinde FIFO okuma disiplinini ihlal eden SDL kavramlarını kapsamaktadır. Geliştirilen yaklaşımın sonuçları POVSDL adlı bir yazılım aracı ile gerçekleştirilmiştir ve deney sonuçları karışıklık azaltımının büyük oranda olduğunu göstermiştir.

Anahtar Kelimeler: doğrulama, kısmi sıralama azaltımı, sürekli küme, biçimsel belirtim, ayrıştırıcı, yazılım mühendisliği, iletişim yazılımı, SDL

ÖZ

AYRIK OLAYLI DİZGELER İÇİN KONTROL İÇERME/SEÇME YAKLAŞIMI VE KATMANLI KONTROLE UYGULANMASI

Nergis, Aydın

Doktora, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Kemal İnan

Ekim 1997, 106 sayfa

Bu tez ayrık olaylı dizgelerin kontrol problemleri için içerme/seçme yaklaşımı olarak adlandırılan yeni bir yaklaşımı önermektedir. Bu yaklaşımda kontrol edilen süreç, içerme süreci olarak adlandırılan tüm olası kontrol davranışları içeren bir süreç olarak tanımlanır. Bu şekilde denetleyici kavramı ortadan kalkar ve kontrol yöntemleri bir içerme sürecinden tam veya eksik tanımlı davranışların seçimine indirgenir. Bu yaklaşım sürekli kontrol dizgeleri için geliştirilen ve 'Differential Inclusions' olarak adlandırılan kuramın ayrık olaylı dizgelerdeki karşılığı olarak görülebilir.

Bu tezde içerme/seçme kuramının fikirleri ve teknikleri katmanlı (hiyerarşik) dizgelere uyarlandı. Üst katmadaki herhangi bir seçme alt katmanda en az bir seçmeye karşılık gelecek şekilde üst katman içerme süreci sistematik olarak alt katman içerme sürecinden elde edildi. Karmaşıklığı azaltmak için kuram, bir bilgisayar programlama dili yorumlayıcının üst seviye bir dilden alt seviye

çıkış dilini üretmesine benzer şekilde üst katman seçme yapı taşlarından alt katmanın tüm seçimlerini üretir.

Anahtar Kelimeler : Ayrık olaylı dizgeler, Denetimli Kontrol, Katmanlı Kontrol

ÖZ

UYUM SINAMASI ÜRETİLMESİ İÇİN ALGORİTMALAR

Serdar, Burak
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Güney Gönenç
Ortak Tez Yöneticisi: Prof. Dr. Kemal İnan

Temmuz 1998, 63 Sayfa

Bu tezde, sonlu durum makinası belirtimleri için uyum sınaması üretilmesi için bir model önerilmiştir. 'Kara kutu' uygulamaların belirtimlerine uyumunu sınamanın tek yolu, uygulamaya girdiler vermek ve beklenen çıktıların alınıp alınmadığını gözlemektir. İdeal olarak, bir uyum sınaması uygulamalardaki olası her hatayı bulmalıdır. Önerilen model, belirli varsayımlar altında tüm hataları bulmak için kontrol dizilerinde bulunması yeterli olan koşulları belirlemekte kullanılmıştır. Ayırt edici diziler ve tek girdi/çıktıdizileri kullanılarak, sıfırlama olduğu ve olmadığ ı durumlar için algoritmalar geliştirilmiştir. Bu algoritmalar C++ programlama dili kullanılarak gerçeklenmiştir.

Anahtar sözcükler: Uyum sınaması, ayırt edici diziler, tek girdi/çıktı dizileri

BİBLİYOGRAFİK BİLGİ FORMU

1. Proje No : EEEAG-COST 247	**2. Rapor tarihi:** 22/10/2000

3. Projenin başlangıç ve bitiş tarihi : 1/3/1994 - 1/10/1997

4. Projenin Adı : Biçimsel yöntemler ile dağılımlı yazılımların doğrulanması ve geçerlileştirilmesi

5. Proje Yürütücüsü ve Yardımcı Araştırıcılar : K. İnan (yürütücü), A. Sulimov, V. Levin, E. Bounimova, H. Yenigün, O. Başbuğoğlu, A. Sezgin, H. Çalgın, O. Gücün, U. Doyuran, C. Tunçtekin, D. Çenberci, İ. Dalkıran, A. Şen, A. Nergis, B.Serdar

6. Projenin Yürütüldüğü Kuruluş ve Adresi : Orta Doğu Teknik Üniversitesi, Balgat-Ankara

7. Destekleyen Kuruluş(ların) adı ve adresi : COST, NATO

8. Özet(Abstract) : Gerçek zaman iletişim yazılımlarında kullanılan dağımlı yazılımların betimlenmesi ve doğrulanması için iki ticari SDL ürünü olan SDT (Telelogic) ve GEODE (Verilog) paketleri kullanılarak uygulamalar yapıldı ve yeni teknikler geliştirildi. Özellikle *Lucent Technologies*'de geliştirilmiş olan COSPAN doğrulayıcısının SDL platformunda kullanılabilmesi için SDL-S/R derleyicisi gerçekleştirildi ve SDL dili ile COSPAN doğrulayıcısının donanım/yazılım entegre tasarımı (*HW/SW co-design*) için genişletilmiş SDL tanımı yapıldı. Bunun yanısıra SDL kullanımında durum patlaması olarak anılan karmaşıklık sorununa çözüm getirmek üzere teknikler geliştirildi. (*In order to specify and validate embedded real-time software used in communication systems the commercial products SDT and GEODE were applied to numerous examples and techniques were developed for that purpose. In particular to use COSPAN verifier developed at Lucent Technologies within an SDL platform a restricted SDL to S/R compiler was implemented and SDL was extended as the first step of implementing a HW/SW codesign tool using the COSPAN verifier. In addition complexity relief techniques were developed to overcome the problem of state explosion when using SDL)*

Anahtar Kelimeler : Biçimsel teknikler, betimleme, doğrulama, SDL, iletişim yazılımı, uyum sınama, donanım/yazılım entegre tasarımı,

9. Proje ile ilgili Yayın/Tebliğlerle ilgili Bilgiler :
Bkz. Kesin Rapor bölüm 3.

10. Bilim Dalı : Bilgisayar mühendisliği **ISIC kodu** :
Doçentlik B. Dalı kodu :
Uzmanlık alanı kodu :

11. Dağıtım : ☐ **Sınırlı** ☑ **Sınırsız**

12. Raporun Gizlilik Durumu : ☐ **Gizli** ☑ **Gizli değil**