N° d'ordre : 1808

THESE

présentée

pour obtenir le titre de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

SPECIALITE: Informatique et Télécommunications

par

M. Iulian Sorin OBER

Spécification et Validation des Systèmes Temporisés avec des Langages de Description Formelle: étude et mise en œuvre

Soutenue le 21 septembre 2001 devant le jury composé de :

M.	**Zoubir Mammeri**	**Président**
M.	**Roland Groz**	**Rapporteur**
M.	**Joseph Sifakis**	**Rapporteur**
M.	**Bernard Coulette**	**Directeur des travaux de recherche**
Mme.	**Susanne Graf**	**Examinateur**
M.	**Alain Kerbrat**	**Examinateur**

Remerciements

Je remercie Bernard Coulette d'avoir accepté d'encadrer cette thèse, ainsi que pour son soutien pendant ces années.

Je remercie Alain Kerbrat d'avoir initié ces travaux, ainsi que pour l'encadrement soutenu et pour nos discussions souvent contradictoires mais toujours enrichissantes.

Je remercie M. Roland Groz et M. Joseph Sifakis d'avoir accepté d'être rapporteurs de cette thèse. L'intérêt avec lequel ils ont lu et commenté mon manuscrit me fait honneur.

Je remercie M. Zoubir Mammeri de m'avoir fait l'honneur de présider ce jury.

Je dois un grand merci à l'équipe du laboratoire Verimag avec qui j'ai collaboré pendant la durée de cette thèse. Merci à Marius Bozga, Susanne Graf et Laurent Mounier pour les discussions très actives, et pour avoir accompagné mes premiers pas dans l'univers de la vérification formelle. Je remercie également Stavros Tripakis pour la patience avec laquelle il a répondu à mes questions. Enfin, je remercie Susanne Graf d'avoir accepté de juger ce travail.

Je remercie la Compagnie des Signaux et la Société Telelogic d'avoir rendu tout cela possible en m'accueillant dans leur centre toulousain. J'ai trouvé là une équipe merveilleuse sur le plan humain, et impressionnante sur le plan technique. En particulier, je tiens à remercier Jean-Luc Roux pour son soutien constant pendant la rédaction de cette thèse. Merci à tous les autres collègues, une liste serait trop longue.

Je tiens à remercier Dan Chiorean pour m'avoir donné l'occasion de faire mes premiers pas en recherche dans un excellent environnement. Je le remercie pour son soutien amical, mais aussi pour les leons spontanées d'esprit critique.

Enfin, je suis reconnaissant envers ma famille et mes amis pour avoir été à mes côtés pendant ces années. Merci Ileana.

Résumé: Ce travail porte sur les techniques de description et de validation d'une catégorie de systèmes temps-réel, dont le comportement est contrôlé ou conditionné par le temps (*systèmes temporisés*). La validation de ces systèmes nécessite la prise en compte simultanée des aspects temporels et comportementaux. Pour cette raison, nous nous intéressons à l'extension des formalismes de description du comportement avec des informations temporelles, et à la mise en oeuvre des techniques d'analyse associées.
Le langage cible choisi est LDS, dont l'usage est très répandu dans l'industrie, et qui présente d'autres avantages: il est standardisé, il a une sémantique formelle, et il bénéficie d'outils avancés de validation par simulation ou vérification. Pour intégrer les extensions temporelles, nous avons pris comme modèle et base sémantique les automates temporisés.
Nous proposons un ensemble d'extensions du langage LDS, qui permet de décrire le comportement dépendant du temps ainsi que les hypothèses temporelles sous lesquelles le système fonctionne. Les extensions sont formalisées en ASM, en complément de la sémantique normalisée de LDS. Nous établissons le lien entre LDS et le modèle des automates temporisés, ce qui nous permet ensuite d'adapter des méthodes d'analyse spécifiques.
Nous étudions également deux langages utilisés pour la description des propriétés des modèles LDS: MSC et GOAL. Dans le cas de GOAL, des extensions sont introduites pour exprimer des propriétés temporelles quantitatives. Dans le cas de MSC, nous proposons une sémantique pour les aspects temporels de MSC-2000. Pour les deux langages, nous étudions des méthodes de vérification par model-checking.
Nous présentons la mise en oeuvre des techniques étudiés dans un outil de simulation et vérification, qui a permis de montrer l'intérêt mais aussi les limites d'utilisation des ces techniques sur un ensemble de cas d'étude.

Mots clé : systèmes temporisés, SDL, MSC, Automates Temporisés, ASM, sémantique opérationnelle, vérification par model checking

Abstract: This work deals with the description and validation of a category of real-time systems, whose behaviour is controlled or conditioned by time (timed systems). The validation of this kind of systems must take into account both behavioural and timing aspects. For this reason, we are interested in extending the behavioural description formalisms with timing information, and in subsequently applying timing analysis techniques.
The study focuses on the SDL language, because it is widespread in the real-time systems industry and presents several other advantages: it is standardised, it has a formal semantics, and benefits from advanced validation tools. We integrate constructs for capturing timing information, as well as timing analysis methods in the framework of SDL, by taking timed automata as model and semantic basis.
We propose a set of extensions of SDL, which allow the description of time-dependent behaviour and of timing hypotheses under which a system works. The extensions are given a formal semantics in ASM, which complements the standard SDL semantics. We also describe the link between SDL and the timed automata model, which allows timed automata analysis techniques to be adapted to SDL.
In this framework, we use two additional languages for expressing quantitative temporal properties related to SDL models: MSC and GOAL. For GOAL, we propose a set of extensions that allow modeling information about time. For MSC, we discuss a timed semantics that encompasses the new timing constructs of MSC-2000. For both languages we examine the problem of property verification by model checking.
The thesis ends with the description of a simulation and verification tool built in the context of this work, and presents some case studies used to validate the proposed concepts and techniques.

Keywords : timed systems, SDL, MSC, Timed Automata, ASM, operational semantics, model checking

Contents

Chapter 1

Résumé

1.1 Introduction

Cette thèse porte sur les techniques de description et de validation du comportement d'une catégorie de systèmes temps-réel, que nous appelons *systèmes temporisés*. Conformément à une définition généralement acceptée, les systèmes temps-réel sont des systèmes dont le fonctionnement correct dépend de la satisfaction de certaines contraintes temporelles. Parmi ces systèmes, nous désignons comme *temporisés* ceux dont le comportement est *contrôlé* ou *conditionné* par le temps, à la différence des systèmes où le temps apparat seulement par l'intermédiaire des facteurs de performance.

Une méthode de validation des systèmes temps-réel doit tenir compte à la fois des contraintes fonctionnelles et des contraintes temporelles appliquées au système. Néanmoins, pour la majorité des systèmes temps-réel, les deux aspects peuvent être traités séparément, par exemple en utilisant des modèles spécifiques pour chacun d'eux. Plusieurs classes de modèles et de méthodes d'analyse, portant sur un aspect particulier du système modélisé, sont habituellement utilisées dans l'ingénierie des systèmes temps-réel. Nous mentionnerons en particulier les *modèles d'ordonnancement* et les *modèles de performance*. Les *modèles d'ordonnancement* (voir la monographie [KRPO93]) visent à étudier les aspects de compétition pour les ressources (y compris le temps de calcul), et peuvent fournir une base pour la validation du comportement temporel d'un système. Cependant, ils ne sont pas applicables aux systèmes complexes dont le fonctionnement dépend du temps (tels que les systèmes temporisés). Les *modèles de performance* visent à décrire les systèmes d'un point de vue probabiliste, et peuvent être utilisés pour la validation des contraintes de performance; cependant, leur capacité à décrire le comportement du système reste limitée.

Dans le cas des systèmes temporisés, examinés dans cette thèse, les aspects comportementaux et les aspects temporels d'un système ne peuvent être séparés, et une méthode de validation doit porter sur un modèle hybride incluant les deux. Pour cette raison, nous nous intéressons aux modèles fonctionnels (comportementaux) d'un système, et à leur extension avec des constructions pour exprimer les informations temporelles. Plus précisément, nous étudions les *méthodes formelles* de description et de validation, s'appuyant sur des langages formalisés tels que LDS [IT99b], et sur des méthodes de validation telles que la simulation ou la vérification par *model-checking* [QS82, CES86].

Le point de départ de cette thèse est l'écart que nous avons constaté entre l'état de l'art et l'état de la pratique industrielle dans le domaine de la spécification et de la vérification des systèmes temporisés. D'une part, plusieurs langages de modélisation, tels que LDS [IT99b],

HRT-HOOD [BW95], ROOM [SGW94] ou UML avec des extensions temps-réel [Dou99, Dou98, SR98], sont utilisés dans l'industrie. Les concepts de modélisation utilisés dans ces langages ont beaucoup évolué, mais les méthodes d'analyse employées dans les outils industriels ne couvrent pas les dernières avancées de la recherche. Du côté de la recherche, il y a beaucoup de modèles abstraits, tels que les automates temporisés [ACD93, AD94], les extensions temporisées des réseaux de Petri [MF76, Sif77, Ram74] ou encore les extensions des algèbres des processus [NS91], qui ont été développés en parallèle avec des méthodes d'analyse telles que la simulation, le model-checking, et la réécriture. Cependant, l'acceptation des modèles, des méthodes et des outils académiques par l'industrie se fait lentement, à cause de leur complexité et du support limité qu'ils offrent pour la conception des systèmes complexes.

En partant de ce constat, l'objectif de la thèse est d'intégrer dans le langage LDS les techniques de modélisation et d'analyse récemment développées dans le domaine des automates temporisés, et d'étendre les outils associés. Nous avons choisi LDS car c'est un langage largement répandu dans l'industrie temps-réel. Les autres avantages de LDS sont le fait qu'il soit standardisé, qu'il bénéficie d'une sémantique formelle, et que les outils de validation basés sur LDS sont plus proches de l'état de l'art dans le domaine. Le choix des automates temporisés comme base théorique pour les extensions temporelles de LDS se justifie par le fait que de nombreux travaux de recherche récents ont concerné ce modèle, et par conséquent beaucoup de problèmes théoriques (liées à la décidabilité du modèle, aux extensions possibles, aux problèmes de model-checking, etc.) ont été étudiés.

1.1.1 Contributions de la thèse

Les résultats de cette thèse peuvent être situés sur trois niveaux:

- Au niveau du langage LDS, avec des propositions d'extension et une sémantique du temps adaptée aux besoins de l'analyse temporisée,
- Au niveau des langages d'expression de propriétés, complémentaires à LDS,
- Au niveau des méthodes et des outils de simulation et de vérification.

Nous présentons les résultats plus en détail dans la suite de cette section.

Extensions du langage LDS

Nous commenons cette thèse avec une description de l'existant, et en particulier nous analysons la faon dont LDS couvre la description des informations temporelles agissant sur le comportement des systèmes modélisés. Nous identifions ainsi certaines lacunes dans la définition de LDS, vis-à-vis des aspects temporels mentionnés.

Ce problème de modélisation est abordé dans le Chapitre 6, où nous proposons une série d'extensions du langage, capables de représenter l'information descriptive sur le temps. Les extensions proposées permettent de décrire un comportement dépendant du temps, ainsi que les hypothèses temporelles sous lesquelles le système fonctionne, ceci directement dans LDS. Ces informations peuvent ensuite être utilisées par des outils d'analyse temporelle, tels que ceux développés dans le cadre de ce travail (présentés plus loin).

Les résultats que nous présentons ici ont été décrits dans nos papiers récents [BGK+00, BGM+01]. Avec un group de partenaires industriels et universitaires, nous sommes en train de raffiner et consolider l'ensemble des extensions de LDS pour le soumettre à l'ITU en vue de leur normalisation.

Sémantique du temps en LDS

La définition standard de LDS comporte une sémantique formelle [IT99c], qui fournit une correspondance entre l'ensemble des spécifications LDS et un ensemble d'objets mathématiques, ainsi qu'une interprétation mathématique de la notion d'exécution d'un système LDS. Dans la deuxième partie du Chapitre 6, nous présentons la sémantique des extensions du langage, ainsi qu'une sémantique appropriée de la notion de temps, dans le même formalisme ASM que [IT99c]. Cette sémantique précise la définition des extensions et fait la liaison avec les méthodes d'analyse utilisées ensuite sur des spécifications étendues.

Spécification des propriétés temporelles quantitatives

L'application des méthodes de validation choisies dans ce travail nécessite un formalisme de description des propriétés temporelles quantitatives. Nous avons considéré pour ce rôle deux langages, couramment utilisés avec LDS: GOAL et MSC.

GOAL [ALH95] est un langage d'observation défini par rapport à LDS et implémenté par l'outil *Object*GEODE [TEL00a]. Dans le Chapitre 7 nous nous intéressons à la description des propriétés temporelles quantitatives avec GOAL, et nous décrivons un ensemble d'extensions de GOAL à cet effet.

Dans le cas de MSC, certaines constructions pour exprimer des contraintes temporelles existent dans la dernière version du langage, MSC-2000 [IT99a], mais ces constructions n'ont pas encore une sémantique formelle. Nous proposons une sémantique temporisée pour un sous-ensemble de MSC-2000, basée sur les automates temporisés. Nous analysons également le problème de la satisfaction d'une propriété MSC par une spécification LDS.

Les deux langages mentionnés ci-dessus sont des langages orientés événement. Pour valider des propriétés décrites avec ces deux langages, il faut d'abord leur donner une base formelle; nous définissons donc un formalisme abstrait orienté événement, basé sur les automates temporisés, que nous appelons *automate de propriétés temporelles* (TPA). Dans le Chapitre 7 nous étudions également les problèmes de vérification (model-checking) posés par l'introduction des TPA.

Méthodes et outils de simulation et de vérification

La dernière partie du travail présenté dans cette thèse concerne les méthodes de simulation et de vérification de spécifications LDS étendues et des propriétés exprimées en GOAL et MSC. Cette partie a abouti à la réalisation d'un outil, dérivé d'un produit industriel (*Object*GEODE). Du point de vue théorique, la partie importante de l'outil est l'algorithme d'exploration symbolique de l'espace d'états d'un système LDS étendu (et des propriétés annexes). Nous définissons l'algorithme par les formules de calcul des successeurs d'un état symbolique. Un algorithme similaire à été présenté antérieurement dans [Boz99]; cependant, à notre connaissance, c'est la première fois que l'ensemble de formules de calcul des successeurs est caractérisé formellement et accompagné d'une preuve mathématique.

1.1.2 Organisation du document

La première partie présente l'état de l'art dans le domaine de la spécification et de la validation des systèmes temporisés. Nous décrivons le langage LDS (Chapitre 3), les langages MSC et GOAL (Chapitre 4), et le modèle des automates temporisés (Chapitre 5).

La deuxième partie présente les extensions temporisées des langages, les méthodes d'analyse et les outils développés dans le cadre de ce travail. Le Chapitre 6 présente les extensions apportées à LDS, et étudie leur impact sur la sémantique formelle de LDS. Le Chapitre 7 introduit le formalisme des TPA's, puis les extensions et la sémantique des langages GOAL et MSC. Le Chapitre 8 porte sur les méthodes d'analyse des spécifications LDS, MSC et GOAL, et sur le développement des outils support.

La partie finale du document contient une étude de cas (Chapitre 9) et les conclusions tirés de ce travail (Chapitre 10).

1.2 Présentation de l'existant

1.2.1 Spécification des systèmes temporisés en LDS

Ce travail débute par une étude de LDS, du point de vue des constructions du langage, et du point de vue de la sémantique. Cette étude s'appuie sur la version '2000 du langage [IT99b], et sur la sémantique formelle de LDS décrite en ASM [IT99c]. L'étude réalisée couvre la majorité des concepts de LDS-2000 (relatifs à l'architecture, à la communication, à la description du comportement, et aux données), ainsi que le formalisme ASM et la sémantique statique et dynamique du langage. Dans la suite du paragraphe, nous allons présenter les conclusions de cette étude concernant l'expression du comportement *temporisé*, et la sémantique du temps, qui sont essentielles pour le reste du travail réalisé dans cette thèse.

Description du comportement temporisé en LDS-2000

LDS définit des constructions qui permettent la description du comportement temporisé. Il existe deux types de données relatifs au temps dans LDS: *Time* et *Duration*. Les valeurs du type *Time* représentent des moments sur une échelle de temps depuis l'initialisation du système, tandis que les valeurs du type *Duration* représentent des distances relatives (différences) entre moments sur l'échelle absolue. Des opérateurs spécifiques sur les valeurs de ces types (addition de temps et de durées, multiplication de durées, etc.) sont prédéfinis dans le langage.

Le temps présent (i.e. écoulé depuis l'initialisation du système) est consultable par l'intermédiaire de l'opérateur prédéfini **now**. La manière dont le temps s'écoule n'est pas définie dans LDS: la seule hypothèse qui est faite sur les valeurs de **now**, est que leur évaluation successive donne toujours des valeurs croissantes.

Ainsi le comportement dépendant du temps peut être décrit des deux manières: soit en utilisant la valeur de **now** dans des tests ou des expressions de tirage de transitions, soit en utilisant des *temporisations*.

Les *temporisations* sont des objets spéciaux du langage LDS, qui ont des attributs spécifiques comme les données (e.g. état d'activité), mais aussi un comportement prédéfini (indépendant du comportement des agents du système LDS). Une *temporisation* peut être définie par un agent LDS (avec le mot clé **timer**); l'agent peut ensuite armer la temporisation avec une date d'échéance (par l'opération **set**), la désarmer (par l'opération **reset**), ou consulter son état (par l'opération **active**). Le comportement d'une temporisation est le suivant: la temporisation est inactive tant qu'elle n'a pas été armée. Une fois que la temporisation est armée, elle devient active et attend l'arrivée de son échéance. Quand l'échéance arrive, la temporisation expire et un *signal est déposé dans la file d'attente* de son agent propriétaire. Si la temporisation est

désarmée avant que l'échéance n'arrive ou avant que le signal correspondant ne soit consommé, elle redevient inactive et tout signal correspondant est effacé de la file d'attente.

Il est important de noter le fait que l'expiration d'une temporisation produit un signal asynchrone, qui passe toujours par la file d'attente de son agent propriétaire. Par conséquent, quand un signal issu d'une temporisation est consommé, la seule hypothèse garantie par la sémantique de LDS est que l'échéance de la temporisation a eu lieu. En principe, on ne peut rien supposer à propos du temps passé depuis l'échéance. Cette hypothèse minimale est correcte du point de vue des implémentations d'un système LDS, mais présente un certain nombre d'inconvénients quand la spécification LDS est utilisée à des fins de simulation ou de vérification.

Description vs. spécification

Comme décrit dans l'introduction de la norme Z.100 [IT99b], LDS vise à la fois la *spécification* de haut niveau, et la programmation (*description*) des systèmes. Les deux objectifs du langage sont parfois conflictuels, et le côté programmation a été prioritaire dans sa définition. Pour cette raison, LDS est un langage de conception assez complet, mais il manque certaines constructions pour la modélisation de haut niveau, nécessaires dans les phases initiales de spécification d'un système. Certaines des extensions proposées dans cette thèse permettent donc la spécification abstraite:

- du comportement des canaux, avec des attributs tels que le taux de perte, les délais minimaux/maximaux, etc.
- des temps d'exécution,
- du comportement (temporisé) de l'environnement du système.

Les constructions proposées seront présentées plus en détail dans la suite du résumé.

Sémantique et raisonnement sur le temps

La sémantique du temps dans LDS est présentée en termes formels dans la thèse. Nous dressons ici un résumé des caractéristiques principales de cette sémantique:

- Les actions individuelles LDS (affectation, envoi de signal, etc.) sont atomiques et prennent un temps nul pour s'exécuter. La granularité de l'atomicité des actions composées et des transitions est l'action individuelle.
- L'évaluation des expressions n'est pas atomique. Par conséquent, la valeur de **now** peut varier pendant l'évaluation d'une expression, ce qui peut influencer sur le résultat de l'expression.
- En général, une quantité non-déterminée de temps peut s'écouler entre l'exécution de deux actions ou de deux transitions
- Une conséquence directe du point antérieur est qu'un message correspondant à une *temporisation* peut ne pas être pris en compte pendant une certaine durée (non-déterminée) après son envoi.

Avec ces hypothèses minimalistes, il est difficile de garantir une propriété sur le comportement d'un système dans le temps. D'autre part, beaucoup de comportements peu réalistes sont

considérés comme acceptables par la sémantique. Ce problème a déjà été signalé par d'autres auteurs [Boz99, MGHS96] et il est examiné en détail dans nos travaux récents [BGK+00, BGM+01].

Le problème signalé ici pose de sérieuses difficultés aux outils de simulation et de vérification basés sur LDS. Une solution souvent employée par les outils est de considérer une sémantique du temps complètement différente de celle de la norme, basée sur des suppositions réductrices, telles que: chaque action prend un temps nul, le temps est contrôlé et s'écoule seulement quand le système n'a rien à exécuter, etc. Cette solution tombe dans l'autre extrême, et peut cacher des scénarios d'exécution réalistes du système.

En complément des extensions du LDS présentées auparavant, nous proposons aussi une sémantique alternative du temps, qui résout les problèmes mentionnés ci-dessus. L'idée de cette sémantique est d'inclure des informations sur le progrès du temps, inspirées du modèle des automates temporisés, dans la spécification LDS, et de les utiliser ensuite pour contrôler le progrès du temps dans la simulation ou la vérification.

1.2.2 Spécification de propriétés en MSC et GOAL

Un aspect central de la technique de validation basée sur les modèles considérée dans cette thèse est la spécification des contraintes et des propriétés sur le modèle LDS. Nous avons choisi deux langages couramment utilisés avec LDS pour la spécification de propriétés: MSC et GOAL. Ces deux langages sont décrits en détail dans le mémoire; nous faisons ici une brève présentation des deux langages, et donnons des conclusions concernant leur utilisation pour la spécification de contraintes temporelles.

MSC

MSC est un langage normalisé par l'ITU (norme Z.120, [IT99a]), utilisé pour représenter des traces d'exécution des systèmes distribués en termes de messages échangés entre les entités du système ou avec l'environnement. Les composantes principales d'une spécification MSC sont les *instances* (représentant des entités ou des groupes d'entités d'un système) et les *messages* (qui peuvent représenter diverses modalités de communication, dépendant du système considéré). D'autres types d'évènements peuvent être spécifiés dans les traces MSC, tels que des évènements concernant les temporisations (**set**, **reset**, **timeout**), des actions ou des conditions (informelles). Le langage a aussi des constructions pour structurer (composer) les spécifications. Les types de composition possibles sont: l'alternative entre plusieurs MSC, la composition parallèle (par entrelacement d'évènements) de plusieurs MSC, la répétition ou l'exécution optionnelle d'une MSC.

La dernière version du langage (MSC-2000) propose plusieurs constructions pour exprimer des conditions sur le temps. On peut essentiellement exprimer des *contraintes relatives*, qui spécifient la durée passée entre deux évènements, et des *contraintes absolues* qui spécifient le moment auquel un évènement peut survenir. Les deux types de contraintes sont spécifiés au moyen d'une limite inférieure et d'une limite supérieure, qui sont soit des valeurs constantes de temps, soit des expressions plus complexes du type **Time**. Dans le deuxième cas, les expressions peuvent porter sur des résultats de *mesures de temps*. On peut mesurer en effet, par des constructions spécifiques de MSC-2000, soit le temps auquel un évènement survient soit le délai relatif entre deux évènements.

Dans notre travail, nous avons jugé suffisantes les constructions proposées dans MSC-2000 pour exprimer des contraintes temporelles. Il existe néanmoins plusieurs raisons pour lesquelles

la notation MSC-2000 ne peut, en l'état actuel, être utilisée pour la spécification et la vérification des propriétés temporelles des systèmes LDS:

- La sémantique de MSC-2000 n'est pas formellement définie dans la norme.
- Il n'y a pas de relation de conformité formellement définie entre des spécifications LDS et des spécifications MSC. Cette relation a été juge en dehors de l'objet de la norme Z.120.
- Certaines des constructions de MSC-2000 (notamment les mesures de temps) sont trop expressives et il n'y a pas de méthode de vérification qui puisse les prendre en compte.

Dans la thèse, nous nous sommes intéressés à ces problèmes, et nous avons proposé des solutions qui sont présentées plus loin.

GOAL

GOAL [ALH95] est un langage d'observation supporté par l'outil *Object*GEODE [TEL00a]. Pour plus de détail sur les langages d'observation le lecteur peut consulter les travaux de [Gro89].

Par définition, GOAL a un domaine d'applicabilité plus limité que les MSC, n'étant pas un langage de haut niveau pour spécifier des propriétés abstraites. Le langage est utilisé pour exprimer et vérifier des propriétés comportementales d'un système LDS, et pour guider le processus de simulation et de vérification.

GOAL est un langage orienté évènements, et basé sur des automates. La spécification d'un observateur GOAL ressemble à une machine à états d'un agent LDS. Les transitions de l'observateur sont tirées par des évènements se produisant dans la spécification LDS associée, qui peuvent être des échanges de messages, la création ou l'arrêt des agents, le tir de certaines transitions, etc. Les états d'un observateur peuvent être de trois types: *succès*, *échec* ou *ordinaire*, et correspondent à la satisfaction ou à la non-satisfaction de la propriété spécifiée par l'observateur.

Dans la thèse, nous avons mis en évidence deux problèmes relatifs à l'utilisation de GOAL comme langage de représentation/validation de propriétés temporelles quantitatives:

- L'absence de constructions permettant l'expression des conditions sur le temps.
- L'absence d'une sémantique temporisée.

Les solutions proposées dans la thèse pour résoudre ces problèmes sont présentées plus loin.

1.2.3 Spécification et vérification avec des automates temporisés

Pour compenser les points faibles de LDS, MSC et GOAL, en termes de sémantique temporisée et de méthodes d'analyse, nous nous intéressons à l'application des techniques d'automates temporisés en conjonction avec ces langages.

Le modèle des automates temporisés est un modèle de machines à états étendu avec des constructions pour la spécification des contraintes temporelles. Les éléments essentiels d'un automate temporisé sont des états discrets, des transitions, et des horloges qui mesurent le temps. Pour faciliter la spécification des contraintes complexes, un automate peut utiliser plusieurs horloges qui avancent toutes à la même vitesse, mais qui peuvent être remises à zéro ou consultées séparément.

Le comportement dépendant du temps est spécifié en imposant aux transitions des gardes qui portent sur les valeurs d'horloges. Le modèle limite les formes acceptées dans ces gardes,

pour préserver la décidabilité du modèle: seules les comparaisons d'horloges avec des constantes (entières) ou les comparaisons de différences de deux horloges avec des constantes sont valides.

La sémantique d'un automate temporisé est donnée par un graphe sémantique en temps continu: l'état dynamique d'un automate comprend un état discret (partie de la spécification de l'automate), et une valeur réelle pour chaque horloge du système. Dynamiquement, un automate peut exécuter deux types de transitions: des transitions discrètes (voir la spécification d'un automate), et des transitions temporelles qui signifient le progrès du temps. Les transitions temporelles ne servent qu'à avancer le temps, i.e. à augmenter (uniformément) la valeur de toutes les horloges. En revanche le temps ne progresse pas durant les transitions discrètes, i.e. la valeur de chaque horloge reste la même ou est remise à zéro (dans le cas d'un reset spécifié sur la transition).

Une exécution d'un automate temporisé est donc une succession (finie ou infinie) de transitions temporelles et de transitions discrètes en alternance. Pour pouvoir modéliser des actions (transitions) qui arrivent à un moment précis, le progrès du temps durant l'exécution est par définition relié à l'exécution de l'automate, au moyen des *conditions de progrès du temps*. Ces conditions dépendent de la valeur d'un attribut de chaque transition de l'automate, appelé *urgence* . L'urgence de chaque transition peut avoir les valeurs suivantes: *eager* , *delayable*, *lazy*. Brièvement, la signification des valeurs d'urgence est la suivante:

- Dès qu'une transition *eager* est tirable, le temps ne peut pas progresser. La transition eager ou toute autre transition discrète tirable doit alors être tirée.
- Quand une transition *delayable* est tirable, elle empêche le temps de progresser au-delà de la borne supérieure de sa garde. Les conditions de progrès du temps sont composées, de faon que la condition la plus restrictive s'applique.
- Les transitions *lazy* n'imposent aucune condition sur le progrès du temps.

Méthodes d'analyse et problèmes décidables

Plusieurs problèmes importants pour la validation du comportement sont décidables sur le modèle d'automates temporisés, et des méthodes d'analyse efficaces sont disponibles. On peut par exemple décider de l'atteignabilité d'un état de l'automate, ce qui implique la décidabilité de la vérification des diverses propriétés d'invariance ou de sûreté. Les problèmes de satisfaction des propriétés spécifiées dans diverses extensions de logiques temporelles (ou dans d'autres formalismes tels que les automates temporisés avec des conditions d'acceptation de Büchi) sont aussi décidables, et il existe des méthodes de vérification concrètes pour ces problèmes.

L'analyse des automates temporisés utilise des abstractions, notamment des représentations symboliques du graphe sémantique d'un automate, pour pallier au fait que le graphe sémantique est habituellement infini et non-dénombrable. Dans le mémoire de thèse, nous présentons deux abstractions, le *graphe de régions* et le *graphe de simulation* qui seront ensuite appliquées à l'analyse du langage LDS étendu.

Le modèle des automates temporisés que nous avons choisi pour notre travail impose des restrictions de modélisation, cependant il donne une limite supérieure de complexité des modèles temporisés analysables, dans le sens où plusieurs extensions de ce modèle ont été étudiées avec des résultats essentiellement négatifs concernant la décidabilité et les méthodes d'analyse applicables (voir dans le mémoire de thèse). Pour cette raison, nous considérons que les restrictions du modèle, qui vont se refléter plus tard au niveau du langage LDS étendu, sont inévitables pour préserver l'analysabilité des spécifications LDS.

Idées sur les extensions temporelles et la sémantique de LDS

Comparé avec LDS, les automates temporisés apportent plusieurs idées qui facilitent la spécification du comportement temporisé et le raisonnement temporisé basé sur le modèle:

- Les horloges et les gardes donnent un moyen flexible d'exprimer des contraintes temporelles complexes. Elles peuvent être introduites comme un complément aux constructions existantes de LDS (**now**, temporisations).
- Les *conditions de progrès du temps* limitent les comportements possibles d'un modèle, et permettent de spécifier quels sont les comportements raisonnables du point de vue temporel. Les urgences peuvent être utilisées pour spécifier des actions qui sont exécutées à un moment précis ou dans un intervalle précis de temps, ce qui n'est pas possible en LDS standard.
- Les conditions sur le temps (i.e. sur les valeurs d'horloges) ont des formes restreintes, ce qui permet l'analyse et la vérification automatique des propriétés. En revanche, en LDS standard la complexité des conditions sur **now** n'est pas contrainte, et il n'existe pas de méthodes d'analyse applicables dans le cas général.

1.3 Extensions des langages et méthodes de validation

1.3.1 Extensions de LDS

Notre travail sur l'extension de LDS comporte principalement deux parties: les extensions des constructions du langage pour décrire des informations temporelles, et l'extension /modification de la sémantique formelle.

Extensions pour la représentation des informations temporelles

Les extensions que nous proposons pour LDS sont en partie des constructions inspirées directement du modèle des automates temporisés, et en partie des extensions de plus haut niveau pour la spécification des informations tels que le temps d'exécution des actions, le temps de transmission des signaux, etc.

Horloges, gardes, urgences. Nous proposons une définition des horloges en tant que mécanisme de base pour mesurer et contraindre la progression du temps. Techniquement, les horloges sont introduites en LDS par l'intermédiaire d'un type de données (`Clock`). Les opérations habituelles (création, reset, comparaison avec un entier, différence de deux `Clocks`) sont définies sur les valeurs de ce type.

Les comparaisons de `Clocks` ou de différences de deux `Clocks` avec un entier peuvent être utilisées dans la spécification de la garde d'un transition LDS (la notion de garde existe déjà dans LDS, avec le mot clé **provided**). De plus, cette extension de LDS permet la spécification d'une *urgence* (*eager*, *delayable* ou *lazy*) pour chaque transition.

Durées d'exécution des actions. Cette extension permet la spécification de durées d'exécution des actions au moyen d'une borne inférieure et d'une borne supérieure. Comme dans le modèle des automates temporisés, les actions LDS s'exécutent dans un temps nul, mais les actions qui prennent du temps sont simulées par un état implicite symbolisant l'action en cours d'exécution, et par une transition *delayable* symbolisant la fin de l'exécution.

La spécification des canaux. Les canaux LDS standards ne perdent jamais de signaux, et les délais appliqués aux signaux transférés sont soit nuls, soit non-spécifiés. Pour valider le comportement d'un spécification LDS standard avec des hypothèses précises sur le comportement des canaux, l'utilisateur doit modifier le modèle LDS et fournir une description impérative des canaux comme agents, avec tous les inconvénients inhérents, qui sont présentés dans la thèse. De plus, à cause de la sémantique non-contrainte du temps dans LDS, le comportement précis des canaux ne peut être garanti.

L'extension proposée dans la thèse permet de spécifier un taux de perte et des bornes minimales et maximales pour les délais appliqués aux signaux transmis sur un canal. On définit deux types de délais pour les canaux: cumulatifs et non-cumulatifs. Dans le cas de délais cumulatifs, les temps d'arrivée des signaux qui précèdent un signal sont rajoutés au temps d'arrivée du signal concerné. Dans le cas de délais non-cumulatifs, le temps d'arrivée d'un signal est compris strictement entre les bornes spécifiées sur le canal, et il est contraint par les signaux précédents seulement par le fait que les canaux sont FIFO. Les canaux non-cumulatifs correspondent aux liaisons qui font un traitement en parallèle (ou en chane) des signaux, tandis que les canaux cumulatifs correspondent aux liaisons où les signaux sont transmis un par un.

Le mémoire de thèse illustre ces concepts sur un exemple réel, le protocole SpaceWire [sWG00] développé par l'Agence Spatiale Européenne, spécifié en LDS avec les extensions décrites ci-dessus.

Sémantique des extensions et du temps

La sémantique des extensions et une nouvelle sémantique du temps sont décrites dans la thèse, en utilisant le formalisme ASM. Les définitions introduites complètent la sémantique standard de LDS [IT99c]. Nous ne reprenons pas ici ces définition, mais nous soulignons les points difficiles et les choix qui ont été faits.

La sémantique des canaux et des temporisations dans LDS (standard) utilise le concept de *schedule*, qui sert pour retarder l'arrivée d'un signal (e.g. un signal correspondant à une temporisation) à son agent de destination. Nous avons étudié deux faons de traiter les temporisations et les canaux à délai (borné) dans LDS étendu: soit en utilisant les *schedules* , soit en utilisant des horloges implicites. La deuxième alternative s'avère préférable, car elle ne s'appuie pas sur une notion de temps absolu (comme c'est le cas dans les *schedules*) mais sur des mesures relatives, ce qui permet l'application des techniques d'analyse d'automates temporisés.

Un point important de la sémantique du temps que nous proposons est la contrôlabilité. En effet, le temps dans la sémantique standard de LDS est considéré comme un paramètre extérieur au système. Cela se reflète dans le fait que **now** est une fonction dite **monitored** dans la sémantique ASM standard. Pour pouvoir introduire des *conditions de progrès du temps* comme dans les automates temporisés, le temps doit être un paramètre contrôlé par le système en fonction des transitions tirables et de leur urgence.

Modifier de cette faon le statut du temps implique des nombreuses transformations dans la sémantique associée. Nous avons introduit un nouvel agent responsable de l'avancement du temps et des valeurs d'horloges, et nous avons décrit les conditions de progrès du temps en ASM, en fonction de l'état des tous les agents LDS et de l'état des canaux à délai. Par le fait qu'une condition de progrès du temps est une condition globale qui porte sur l'état des toutes les composantes d'un système, nous avons été obligés d'introduire des synchronisations supplémentaires entre les agents ASM définis par la sémantique, qui sinon sont entièrement asynchrones.

Correspondance avec les automates temporisés

Les outils de simulation et de vérification existants basés sur LDS n'utilisent pas la sémantique ASM, mais construisent directement un graphe d'états global du système en utilisant des simplifications pour des raisons d'efficacité. Cette notion de graphe d'états est en fait assez proche du modèle sémantique des automates temporisés, et nous pouvons l'étendre pour y inclure les extensions proposés pour LDS (notamment les horloges). Le résultat est un graphe d'états en temps continu qui ressemble beaucoup à celui des automates temporisés, et sur lequel nous pouvons appliquer des techniques d'analyse spécifiques. Les composantes – états et types de transitions – de ce graphe sont décrites dans le Chapitre 6 du mémoire.

1.3.2 Description et vérification des propriétés temporisées avec MSC et GOAL

Nous nous sommes intéressés aux problèmes décrits dans la section 1.2.2, qui empêchent l'utilisation de MSC et de GOAL en tant que langages de propriétés temporisées pour les systèmes LDS. Les travaux que nous avons réalisés concernent trois niveaux: les constructions introduites dans les langages, leur sémantique, et les méthodes de vérification automatique.

Automates de Propriétés Temporisés

Pour donner une base sémantique solide aux deux langages, nous avons défini un modèle abstrait de description de propriétés, calqué sur les automates temporisés, que nous appelons Automates de Propriétés Temporisés (TPA).

Un TPA est un automate temporisé équipé d'une condition d'acceptation de type Büchi. La différence entre les TPA et les variantes d'automates temporisés de Büchi (TBA) proposés dans [Alu91, Tri98] est que le modèle des TPA est orienté événement et non pas orienté état. Cela signifie qu'une propriété TPA porte sur les évènements qui on lieu dans le modèle associé, et non pas sur les états du modèle. Cette différence est importante, dans la mesure où MSC et GOAL sont tous les deux des langages orientés évènement.

Dans le mémoire nous décrivons formellement le modèle des TPA, et la relation de satisfaction entre un automate temporisé et un TPA. Nous étudions aussi le problème de la vérification de la satisfaction, et nous proposons un algorithme basé sur l'utilisation du *graphe de simulation* des automates temporisés. Cette méthode d'analyse est utilisée ensuite pour vérifier des propriétés MSC et GOAL sur des systèmes LDS étendus.

MSC

Au niveau des MSC, les deux problèmes principaux sont l'absence d'une sémantique qui prenne en compte les aspects temporels du langage, et l'absence d'une relation de satisfaction entre des spécifications LDS et des propriétés MSC.

Nous proposons une sémantique temporisée basée sur les automates temporisés, en partant de la sémantique non-temporisée basée sur des réseaux de Petri proposée dans [GPR93] et en l'étendant avec des horloges et des contraintes temporelles. Nous arrivons ainsi à traiter la plupart des contraintes exprimables en MSC-2000. Les parties du langage pour lesquelles nous ne pouvons pas donner une sémantique concernent notamment les *mesures de temps*, et l'utilisation des variables ou paramètres de type **Time**.

La composition séquentielle des MSC pose aussi des problèmes de décidabilité, déjà signalés par d'autres auteurs [MP00]. Pour pouvoir utiliser les MSC dans la vérification de propriétés, nous avons restreint la définition de la composition séquentielle, de faon à ce que le langage de traces généré par un MSC composite soit toujours régulier.

Nous proposons aussi des définitions possibles pour la relation de satisfaction entre des spécifications LDS et des propriétés MSC. En interprétant l'automate temporisé qui donne la sémantique d'un MSC comme un TPA, nous avons trouvé des correspondances entre la satisfaction des MSC et la satisfaction des TPA. Par conséquent, nous pouvons donner une méthode concrète de vérification pour les propriétés MSC, en utilisant les mêmes techniques que dans le cas des TPA.

Des travaux effectués par d'autres auteurs visent aussi à utiliser les MSC en tant que langage des propriétés, pour la vérification formelle. On notera principalement les travaux sur les Live Sequence Charts (LSC, [DH98]), mais aussi les approches proposés par des outils industriels tels que *Object*GEODE [TEL00a]. Cependant, aucun des travaux sur le sujet ne traite à notre connaissance de la partie concernant le temps.

GOAL

Dans le cas de GOAL, la notion de satisfaction et la sémantique du langage sont déjà définies dans l'outil *Object*GEODE. Le langage manque cependant de constructions pour exprimer des contraintes sur le temps, et sa sémantique doit être adaptée à la nouvelle sémantique temporisée de LDS définie dans cette thèse.

Comme constructions temporelles, nous avons proposé des concepts directement inspirés des TPA: des horloges et de gardes. La sémantique de GOAL est relativement facile à définir en termes de TPAs, du fait qu'il y a une relation directe entre les concepts des deux modèles. Même si les extensions de GOAL sont très légères, les études de cas que nous avons effectuées montrent que le langage résultant est très flexible et permet la spécification de propriétés linéaires complexes.

1.3.3 Simulation et vérification temporelles de LDS

Un des objectifs des extensions décrites dans les sections précédentes est de pouvoir valider le comportement temporel des systèmes LDS, par simulation ou par vérification de propriétés. Dans cette section nous décrivons un outil que nous avons développé à cette fin, qui se présente comme une extension de l'outil de simulation et de vérification de *Object*GEODE [TEL00a], dont il réutilise l'architecture globale et les fonctionnalités principales. L'avantage de réutiliser un environnement industriel est que l'implémentation des constructions des langages qui ne sont pas affectées par les extensions temporelles, est obtenue sans effort supplémentaire.

Fonctionnalités et architecture de l'outil

L'outil offre des fonctionnalités pour:

- La *simulation interactive ou aléatoire*. Les fonctionnalités offertes dans ce mode de fonctionnement ressemblent à celles des débuggeurs pour les langages de programmation: exécution pas à pas, conditions d'arrêt, inspection des données. Il existe aussi d'autres fonctions spécifiques: exécution inversée, sauvegarde des scénarios d'exécution, stimulation automatique des modèles ouverts, production des traces sous forme de MSC, analyse de couverture du modèle, etc.

– La *vérification par exploration exhaustive* de l'espace d'états du modèle. L'outil peut vérifier: l'absence de blocages, l'invariance de certaines conditions logiques, l'absence de certaines erreurs dynamiques (e.g. signaux non-attendus), et la satisfaction de propriétés écrites en MSC ou GOAL.

Les deux modes d'utilisation sont basés sur la construction de l'espace d'états du modèle, mais le processus de construction et la taille de l'espace sont différents dans chaque cas. La vérification d'un modèle est toujours effectuée à la volée, et par conséquent l'espace d'états n'est entièrement construit que dans certains cas.

L'outil est formé de deux modules principaux: un *compilateur* des modèles, et une *librairie* englobant les fonctionnalités génériques des simulateurs. Le compilateur prend en entrée un modèle LDS et une ou plusieurs propriétés MSC ou GOAL; il les transforme dans un format exécutable, où les transitions LDS, par exemple, deviennent des routines utilisant des primitives qui implémentent les types d'actions définis dans LDS. Ces primitives font partie de la *librairie*, qui englobe aussi des structures de données standard, et des fonctionnalités génériques (parcours de l'espace d'états, configuration du modèle, etc.).

Au final, le compilateur génère un simulateur (sous forme d'un exécutable séparé) pour chaque modèle LDS. Le simulateur construit l'espace d'états du modèle et implémente toutes les fonctionnalités de simulation et de vérification décrites auparavant.

La construction du graphe de simulation temporisé

L'espace d'états construit par l'outil est une abstraction de l'espace d'états en temps continu défini par la sémantique de LDS. Les états manipulés par le simulateur sont des états symboliques (q, S), où q est un état discret global du modèle (n'incluant aucune information sur le temps et les horloges). S est une *zone* de valeurs d'horloges atteignables dans l'état q, qui a la forme d'un polyèdre (éventuellement non-convexe et non-borné) dans l'espace des valeurs d'horloges ($\mathbb{R}^n$, où n est le nombre d'horloges actives dans q).

Les transitions de ce graphe de simulation correspondent uniquement aux transitions discrètes définies par la sémantique de LDS étendu, qui sont soit des transitions LDS *explicites*, soit des transitions *implicites* (expiration de temporisations, arrivée de signaux retardés sur les canaux). Pour plus de détail, le lecteur peut consulter la sémantique détaillée dans le mémoire. Le calcul des successeurs d'un état (q, S) après l'exécution d'une transition e se fait en deux étapes. On calcule d'abord les états directement atteignables en exécutant la transition e sur chaque état explicite contenu dans l'état symbolique (q, S). On obtient ainsi un autre état symbolique (q', S'). A partir de ce dernier, on calcule combien de temps on peut rester dans chaque état explicite contenu dans (q', S'), et on obtient ainsi l'état symbolique de destination (q', S'').

Les deux étapes décrites ci-dessus sont appelées respectivement le *calcul des successeurs discrets* et le *calcul des successeurs temporels*. Le calcul des successeurs est plus compliqué dans le cas où des propriétés GOAL ou MSC sont associées au modèle LDS, car il faut tenir compte de l'état des automates représentant ces propriétés. En particulier, il peut arriver qu'un état ait plusieurs successeurs différents par la même transition e, du fait que des conditions différentes contenues dans la propriété soient satisfaites par des parties différentes d'un état symbolique.

L'algorithme de calcul des successeurs est présenté en détail dans le mémoire. Il s'appuie sur une représentation de données spécifique (plus particulièrement sur la représentation des polyèdres S par des matrices de différences bornées – DBM [Dil89, ACD93]) et sur des formules de calcul des successeurs temporels qui réalisent des opérations élémentaires sur des polyèdres.

Une partie essentielle du travail effectué est la preuve de la validité de ces formules, présentée dans l'Annexe B.

1.4 Application et conclusion

Nous avons validé les concepts et les outils développés dans le cadre de ce travail sur un ensemble d'études de cas, incluant des cas d'école: un système de barrière de voie ferrée (utilisé précédemment dans [Alu91] et [Tri98]), un protocole de contrôle de flot (BRP, étudié aussi dans [GvdP96, HS96, Mat96, DKRT97]). Ces études de cas ont donné des bons résultats quant à l'expressivité des extensions et à la puissance des méthodes d'analyse utilisées.

Nous avons considéré aussi des études de cas plus complexes. Dans la thèse, nous présentons l'exemple d'un protocole de liaison de données (SpaceWire, [sWG00]) ; l'approche est aussi expérimentée dans le cadre d'autres projets R&D en cours, sur le protocole de multicast RMTP-2 [PMR+00, WPT99] et sur un protocole de synchronisation de flots multimédias.Ces exemples ont montré tout le bénéfice de l'approche retenue, mais aussi certaines limites des extensions que nous proposons, telles que l'impossibilité d'utiliser des mesures de temps dans des systèmes adaptatifs (e.g. contraintes de temps variables en fonction de l'évolution du système).

L'exemple présenté dans le Chapitre 9 du mémoire montre en détail la modélisation des contraintes temporelles contenues dans la norme SpaceWire avec les extensions LDS que nous avons proposées. La validation du modèle du point de vue temporel est aussi discutée, et nous montrons comment la simulation peut être exploitée pour faire des mesures de temps globales qui seront ensuite utilisées pour écrire et vérifier des propriétés en MSC et GOAL.

En conclusion, nous avons développé un ensemble d'extensions, des méthodes d'analyse et un outil support, permettant la description et la validation des systèmes temps réel avec des contraintes temporelles complexes, exprimables dans le langage LDS et les langages connexes MSC et GOAL. Les études de cas réalisées montrent que les extensions proposées sont flexibles et intuitives, et que l'expression des propriétés pour la vérification est aisée, en comparaison avec des langages mathématiques tels que les logiques temporelles. Les méthodes d'analyse développées permettent la dérivation des informations temporelles pertinentes telles que les délais minimaux/maximaux entre évènements.

Dans une perspective future, ce travail peut être continué par la recherche d'un ensemble de concepts de plus haut niveau à intégrer dans les langages de modélisation, éventuellement basés sur les concepts sémantiques proposés ici, qui peuvent apparatre de trop bas niveau et compliquer la conception des modèles. D'autres axes de recherche concernent l'amélioration des techniques de vérification (e.g. par application des méthodes de réduction de l'espace d'états), l'application des nouvelles méthodes de validation (e.g. génération automatique de tests) ou l'intégration dans de nouveaux langages tels que la notation UML.

Chapter 2

Introduction

2.1 Specification and validation of timed systems

This thesis focuses on the techniques for describing and validating the behavior of a class of real-time systems, called *timed* systems. According to a commonly accepted definition [Loc98, HP88, Per90], real-time systems are systems in which correct functioning depends on meeting time constraints. By *timed* systems we designate the class of real-time systems whose functioning is *controlled* or *conditioned* by time. In this way, we differentiate timed systems from other real-time systems in which time appears only as a performance aspect (e.g. through task deadlines, execution times, event arrival times, etc.).

A validation method for real-time systems must take into account both functional requirements and time requirements. For most real-time systems, these aspects may be handled independently, for example using different models of the system (some types of models are briefly presented later in this section). However, in the case of timed systems the behavioral and timing aspects may not be separated, and validation methods must work on hybrid models capturing both facets.

There are several classes of models (and associated analysis techniques) used for the description and validation of real-time systems in general, each concentrating on a different aspect of the system under modeling. *Scheduling models* (see the monograph [KRPO93]) are traditionally associated with the domain of real-time system engineering. They concentrate on the problem of resource contention in real-time systems, disregarding behavioral aspects. Computation time, regarded as a resource, is taken into account by these models; consequently, scheduling models can sometimes constitute the basis for temporal correctness proofs. However, such models are usually ineffective for complex systems whose behavior depends on time (*timed* systems).

Another category of models used in real-time system engineering are *performance models* [Kan92], which give a probabilistic description of a system. Such models include information about the probabilities of discrete events, as well as time-related information. Performance models may be used for validating (statistical) performance requirements; however, as in the case of scheduling models, they do not provide a functional view of a system, and therefore cannot be used for validating combined functional and timing requirements of timed systems.

In this thesis, we concentrate on a class of models that we call *behavioral models*. They describe a system from the functional (computational) point of view, and may additionally contain information about timing. More precisely, we are interested in the application of *formal methods* for describing timed systems, and for validating combined functional and timing properties of systems.

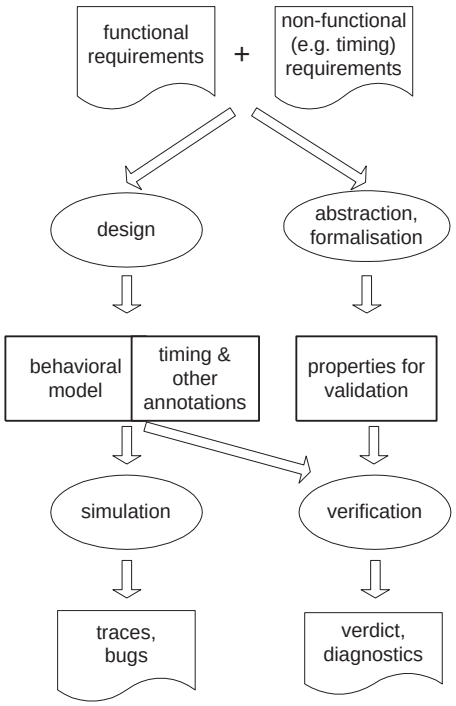


Figure 2.1: Behavioral specification and validation

The classical approach for system specification and validation using behavioral models is represented in Fig. 2.1 and briefly outlined in the following. The starting point of this process are the system requirements, which may be functional requirements (stating the functions the system must fulfill) or non-functional requirements (stating auxiliary requirements such as throughput, quality of service, etc.). A system model is built from these specifications through a design process which usually requires human intervention, and which is normally organized according to a methodology.

The scheme described above may apply to other types of models, not only to behavioral models. The specificity of the latter is that they make a complete description of the system functionality, and thus may be used for several purposes: model-based validation, code generation, testing.

For the model-based validation, another process takes place in parallel with the system design: the formalization of the system properties. The inputs of this process are again the requirements; the outputs are the properties, which are more abstract and concise than the system model, and usually expressed in a specific language.

The validation phase that follows may involve several activities; in Fig. 2.1 we have represented two model-based validation methods, on which we focus throughout this thesis: simulation and property verification. In this work we have left aside other types of validation activities, such as testing. The simulation and verification approaches considered here have in common the fact that both are based on the construction of an abstract semantic model of the system (and of the properties). By simulation we understand a user-guided exploration of this semantic model, in a manner similar to program debugging. Verification is the process through which is is formally proved that the model satisfies the properties extracted from the requirements.

Among the verification approaches that may be found in the literature, in this work we consider *model checking* [QS82, CES86].

Regarding the application of the above scheme in the specification and validation of *timed* systems, we have noticed a manifest discrepancy between the current industrial practice and the state of the art in research. On the side of practice, there are the real-time system modeling languages used in the industry – SDL [IT99b], HRT-HOOD [BW95], ROOM [SGW94], real-time extensions of UML [Dou99, Dou98, SR98], etc. – in which modeling is the primary concern. These formalisms have constantly evolved and use modern design concepts; however, the analysis (validation) techniques used in connection with them in industrial tools have not evolved at the same pace, and do not encompass the latest advancements in research.

On the research side, more abstract models (such as timed automata [ACD93, AD94], timed extensions of Petri Nets [MF76, Sif77, Ram74], timed extensions of process algebras [NS91], or timed extensions of Hoare logic [Sha95, CHR92]) have been developed, in parallel with advanced analysis methods based on techniques like model checking, simulation, rewriting, etc. Although they provide powerful features, models and tools developed in the academia are less appealing to the industrial user because of their complexity and their limited support for modern design features.

Starting from these facts, the objective of this thesis is to integrate a set of modeling techniques and analysis methods recently developed in the field of *timed automata*, within the framework of SDL.

The reasons for choosing these two frameworks are manifold. On one side, SDL is widespread in the real-time systems industry; it is also a standard language, with a sound semantic basis. Finally, existing validation tools for SDL are closer to the state of the art in validation techniques, compared to the tools that work with more informal notations such as UML and ROOM. Timed automata [ACD93, AD94], on the other side, have been the subject of consistent research in the recent years. As a result, many problems concerning timed automata have been studied and given a solution: there are abstractions and algorithms for solving several model checking problems (a recent synthesis can be found in [Tri98]), many extensions of the timed automata model have been studied and the decidability limits are known [HKPV98].

The issues mentioned previously are treated more in-depth in the first part of this thesis, which discusses the state of the art in the specification and validation of timed systems.

2.2 The approach and contribution of the thesis

The goal of this work is to improve the support offered by the SDL language for the abstract modeling of timed systems, as well as for validation of timed system specifications. For that, we have followed two main directions: one concerns the improvement of the SDL language definition, the other concerns the application of timing analysis techniques developed within the framework of timed automata to SDL. The results of this work are outlined in the following.

SDL language extensions. The SDL language definition [IT99b] presents the language as a formalism for both abstract *specification* and complete *description* of system structure and behavior. A closer look at the definition of SDL, however, shows that the programming side has been given priority, to the detriment of abstract, non-programatic modeling. This makes SDL interesting as a *design language*, but provides insufficient support for requirements capturing and abstract modeling.

We approach this problem in the first part of Chapter 6, and propose a series of language extensions which are necessary to capture descriptive information (notably timing information) in the initial phases of system modeling. The language primitives introduced in this part allow the modeling of time-dependent behavior, as well as the introduction of annotations describing the (timing) assumptions under which the system is functioning, directly in the SDL model. This information may subsequently be used by timing analysis tools, such as those built in the context of this work.

The results obtained in this part of the thesis are presented, under a slightly different form, in [BGK+00, BGM+01]. Together with a group of partners from the industry and the academia, we are currently working on a submission to the ITU–T standardization body, which proposes a set of higher-level modeling constructs based on the primitives presented in Chapter 6 to be included in the SDL standard.

Semantics of time in SDL. The standard definition of SDL [IT99b] includes a formal semantics, which maps the set of SDL system specifications onto a set of mathematical objects, and provides a formal interpretation for the notion of system execution. We were interested in the impact of the extensions on the standard formal semantics of SDL, and notably in the aspects which concern the handling of time, in order to complete the definition of the extensions mentioned in the previous paragraph, but also in order to make the connection with the timing analysis techniques that we apply subsequently. The result, presented in the second part of Chapter 6, is a set of mathematical definitions that complement the standard formal semantics of SDL [IT99c].

Timed property specification. The next step towards applying timing property validation techniques on SDL models is the definition of a language for expressing combined functional and timing properties. In this work we have considered two languages that have previously been used for expressing functional properties in connection with SDL models: GOAL and MSC.

GOAL [ALH95] is an observer language used by the *Object*GEODE simulation and verification tool [TEL00a]. A synthetic work on the use of observers for expressing and verifying functional properties of systems may be found in [Gro89], where a precursor of the GOAL language is also defined. In Chapter 7 we discuss a set of simple extensions which allow GOAL to express quantitative temporal properties of SDL models. The results of this work were also presented in [OK01].

In the same chapter, we discuss the possibility of using MSC-2000 for expressing quantitative temporal properties of SDL models. MSC-2000 already contains a number of constructs for capturing timing information, but lacks a formal semantic definition including the timing aspects, essential for using the language in verification. Moreover, many model checking problems are known to be undecidable even on the untimed restriction of the language.

The solution we propose in Chapter 7 for the problems enumerated above includes a restriction of the MSC-2000 language, which diminishes the expressive power of the language but renders it decidable. For this restriction of the language, we sketch a semantics based on timed automata, which covers the timing aspects recently added to MSC-2000. We also describe some possible alternative definitions of the satisfaction relationship between SDL models and MSC-2000 specifications, which can be used in verification.

Both GOAL and MSC-2000 are event-based languages, in the sense that they describe properties in terms of events happening in the associated SDL model. Since timed automata-oriented property specification languages (such as timed extensions of temporal logics, or timed

Büchi automata) are state-oriented, we found it necessary to define an event-oriented property formalism at the level of timed automata, in order to provide a sound semantic basis for MSC-2000 and GOAL as property languages. This formalism is defined in Chapter 7 together with two types of satisfaction relations. The model checking problem for these satisfaction relations is studied thereafter.

Simulation and verification – methods and tools. The final part of this work is concerned with the simulation and verification methods applicable to the extended variant of SDL and to MSC and GOAL properties. This part of the work is materialized in a simulation and verification tool, which is derived from a commercial SDL tool (*Object*GEODE, [TEL00a]).

From the theoretical point of view, the important part in this tool is the symbolic state space exploration algorithm. The algorithm uses an abstraction similar to the *simulation graph* of timed automata defined in [Tri98], and is adapted for exploring simultaneously the SDL model and the associated GOAL and MSC properties. An important part of the state space exploration algorithm consists in the steps for computing:

- the *discrete* successors of a state, in the presence of clock operations such as reset, assignment, creation and deletion.
- the *temporal* successors of a state, in the presence of *urgency*.

Although a similar algorithm has been described previously in [Boz99] in the case of a formalism close to SDL (IF, see also [BFG+99]), this is to our knowledge the first time when a precise characterization of the successors computation formulas is done for a formalism based on timed automata with urgencies, and is accompanied by a correctness proof.

2.3 Organization of the document

This document is structured in three parts.

The first part presents the state of the art in the specification and validation of timed systems. It includes three chapters which present respectively: the language for system modeling used in this work – SDL (Chapter 3), the requirements specification languages MSC and GOAL (Chapter 4), and the timed automata model which provides the theoretical foundations for the timing analysis methods considered in this work (Chapter 5). The chapters are relatively self-contained, and present their respective subject both through its definition and from the point of view of the usage that can be made of it. In the end of each chapter, we make a synthesis of the similar languages/models that can be found in the literature, and we attempt to justify the choice made in this work.

The second part presents the language extensions, the analysis methods and the tools that have been developed in the context of this work. Chapter 6 discusses the extensions proposed for SDL, and their impact on the semantic definition of the language. Chapter 7 begins with the definition of an abstract timed property specification formalism, defined at the level of timed automata, called timed property automata (TPA). The TPA model forms the semantic basis for the definition of GOAL and MSC as timed property description languages, which is done in the rest of the chapter. Chapter 8 closes this part with a description of the simulation and verification algorithms and tools used in connection with the extended SDL, MSC and GOAL languages. Knowledge of the subjects presented in the first part is necessary for understanding the three chapters of this second part.

The final part of this document presents a case study (Chapter 9) on which we have validated the concepts proposed in this work. Chapter 10 draws the conclusions of the work carried out in this thesis, and presents further work directions.

Part I

Languages and Models for Real-Time Systems

Chapter 3

SDL

Specification and Description Language (SDL) is a formal modeling language intended for the specification and description of telecommunication systems. SDL is issued and maintained by the International Telecommunication Union – Telecommunication Standardization Sector (ITU–T), as the Recommendation Z.100 [IT99b].

The efforts for defining a specification and description language for telecommunication systems in ITU begun in the early 1970's, as the field of telecommunication systems engineering was experiencing a paradigm shift from the age of simple electromechanical devices to the age of computer-driven telecommunication devices. The language was designed in order to cope with the multiplying number of services supported by these systems, and the increasing complexity of signaling protocols supporting these services.

The first official version of the language was issued by the ITU–T (CCITT[1] at the time) as Recommendation Z.100 in 1976. It contained several pages of standardized graphical symbols for representing event-action models, with only an implied background of Finite State Machines. The language was further refined until its definition reached a stable form in the 1988 version of the Recommendation Z.100. This version included many of the features still present in the current version of the language, among which: hierarchical architecture modeling concepts, asynchronous communication, a data type system, and extended finite state machines for describing behavior. The language definition also included a formal semantics written in META-IV [ISO96].

The language is maintained on a four-year basis by the ITU–T. Major revisions were issued in 1992 and 2000. The 1992 version added type-based modeling and object-oriented constructs to SDL. The 2000 version introduced several implementation-oriented constructs, with the aim of improving the coverage of all system development phases, from analysis down to implementation. The 2000 version also added new modeling constructs inspired from modern object-oriented modeling languages like UML [OMG99], made some steps towards simplifying the language by removing unused or redundant concepts, and completely redefined the formal semantics of the language using a new underlying formalism (ASM [Gur88, Gur95, Gur97]).

Several bibliographic sources provide a detailed description of SDL. The authoritative source concerning the language itself is the ITU–T Recommendation Z.100 [IT99b]. The 1988 and 1996 versions of the language are described in [BHS91] and respectively [EHS97], with examples and an emphasis on the specification of protocol stacks. [SSR89] describes in more detail SDL-88, and includes some general modeling guidelines. This book has been revised for SDL-92 and the sections on system engineering have been improved, resulting in a new book [OFMP+94].

[1] Comité Consultatif International Téléphonique et Télégraphique

[Mam00] is the most recent book on SDL up to date, containing references to SDL-96[2].

The present chapter describes the main features of the SDL language, as defined in the latest revision of the Recommendation Z.100, [IT99b], on which most of our subsequent work is based. However, as tools implementing SDL-2000 are not available yet, the tools we developed in the context of this work are based on SDL-96. For this reason, throughout this chapter we point out the differences between the two language versions.

In §3.1 we make some general remarks about the applicability domain and the modeling paradigm of SDL. §3.2 introduces the modeling concepts and constructs of SDL. §3.3 discusses the semantics of SDL, with an emphasis on the time and concurrency aspects necessary for understanding the rest of the thesis. In §3.4 we take a look at the existing types of tools for analyzing and exploiting SDL models. Finally, in §3.5 we outline some problematic language issues, which constitute the starting point for a part of the work presented in this thesis.

3.1 Scope and paradigm

The scope of SDL, as defined by the Z.100 Recommendation, is the specification and description of telecommunication systems. The meaning of *specification* and *description* in [IT99b] is:

- the *specification* of a system is an abstract description of its required behavior.
- the *description* of a system is the description of its actual behavior, that is an executable model.

In practice, the use of SDL covers several phases from the system development cycle. Also, the modeling concepts provided by SDL can be used for modeling other types of systems besides telecommunication systems. We paraphrase below a generic characterization of the applicability domain of SDL given in [OFMP+94]. SDL is suited for the description of *discrete reactive* systems, which are systems characterized by an intensive and discrete communication with their environment. Both characteristics mentioned above are important:

- *Reactiveness* characterizes systems in which an execution is not pre-determined by some finite amount of initial data coming from the environment. Instead, the system interacts with an evolving environment, and performs tasks in response to stimuli coming from it. The dominant part of the behavior of a reactive system deals with the interactions and not with internal computation.

  The opposite of reactive systems are *transformational* systems. In transformational systems, internal computations take up a more important part of system behavior. The example provided in [OFMP+94] for the reactive/transformational dichotomy is a telephony switching system versus a meteorological forecast system.

  A telephone switch must permanently monitor the status of the connected lines and react to the requests initiated by phone terminals, switch operators, etc. The reactions are usually not complex from a computational point of view; the complexity of such systems is generated by the large number of parallel components involved, the quantity of services they provide, and the possible interleaving of the requests.

  On the other hand, a weather forecast system takes up the initial meteorological observation data, and performs complex computations in order to obtain a forecast. During

[2] SDL-96 contains only minor revisions to SDL-92; the core of the language remained unchanged between the two versions.

computation, the interaction with the environment is minimal and or not relevant compared to the internal system activity.

SDL is appropriate for describing *reactive* systems, as the behavior of SDL system components is given in terms of stable states and responses to stimuli. Data types, operators and procedures written in an algorithmic language which is part of SDL, can be used to specify the transformational aspects of behavior.

– *Discreteness* characterizes systems in which interaction between components or with the environment is materialized through discrete events. Such discrete events are represented in SDL by the *signal* concept.

  The opposite of discrete systems are *continuous* systems, in which the signals by which system components interact can be modeled as (continuous) functions over a dense time domain. On the lowest level of abstraction, most electronic devices exhibit a continuous behavior. SDL is not suitable for representing system at this level of abstraction; other languages and models exist for this purpose.

SDL can capture *functional* information about a system both at an abstract level using descriptive constructs *(specification* level, in Z.100 terminology) and at a detailed level using imperative constructs *(description* level, in Z.100 terminology). As such, SDL models can be employed in different phases of system development:

1. *Analysis/specification.* In this phase, abstract SDL models focusing on the functionality provided by the system are built. Over-specification can be avoided by using informal action specifications, allowed in SDL.
2. *Design.* SDL design models add details on the architecture of the system, and on the relation between functionality and architecture. The description of functionality can also be refined.
3. *Implementation.* SDL provides imperative programming constructs comparable to those of common programming languages. Additionally, the combined use of SDL and other programming languages and libraries is supported by the standard Z.100 and by most SDL tools.

   Implementation in SDL follows a different paradigm compared to common procedural or object-oriented languages, by supporting parallelism and communication natively, and a stimulus-response description of component behavior.
4. *Validation.* Validation of a system model can take many forms depending on the properties that need to be ensured about the system. Since SDL is used for describing *functional* aspects of a system, SDL models are especially suited for the validation of functional properties.

   Verification and testing are two examples of functional validation methods in which SDL models can be used. Verification is a way of formally validating a property, by using a method of formal reasoning about the SDL model. Verification supposes the existence of a mathematical definition for the models built with SDL. Such a mathematical definition is given by SDL's *formal semantics* [IT99c].

   Testing is performed directly on the implementation, by checking on a set of chosen system executions that the system performs as expected. The SDL model of a system may be used in this case to derive tests (manually or automatically).

Validation of timing aspects of the functioning of real-time systems modeled in SDL is the central theme of the present work.

SDL models have also been used for validating non-functional properties of systems, such as performance properties [DHHMC95b, MT00]. However, since a standard SDL model does not contain the necessary information in order to derive performance parameters, it must be extended with constructs specific to performance models. Such an approach has the advantage of reducing the redundancy that otherwise exists between functional models and performance models, but also the risk of increasing the complexity of the SDL model.

5. *Documentation.* The SDL language provides an easily readable graphical representation which can be used as such for documenting the architecture and functioning of a system.

The support for various development phases is not only a language issue, but also a tool issue. The types of SDL tools supporting the activities enumerated above are discussed in §3.4.

3.2 Language concepts

3.2.1 Language definition artifacts

The SDL language definition [IT99b, IT99c] includes the syntax, an informal semantics written in English, and a formal semantics. The syntax has three variants:

- an *abstract syntax*, which abstracts away from keywords, separators and other tokens, and only gives the relations between language objects. For example, the abstract syntax for a *channel definition*[3] is:

  Channel-definition :: *Channel-name*
  [**nodelay**]
  Channel-path-**set**

  The definition specifies that a channel is defined by a name, an optional **nodelay** attribute, and a *set of* channel paths. The channel paths represent the directions in which the channel conveys messages. There can be at most two *Channel-path* objects in a channel definition; this type of constraint is written in English in the abstract grammar section.

  The meta-language used for describing the abstract grammar is a subset of Meta IV[4] [ISO96]. The definitions resemble usual BNF grammar productions, and use operators such as "`*`","`+`","`|`" and "`[ ]`", as well as the "***-set***" operator yielding an unordered collection of objects.

- a *concrete textual grammar* given in extended BNF. The relation between abstract grammar nodes and concrete grammar non-terminals is described in the text.

- a *concrete graphical grammar* which specifies in a *formal* way the contents of SDL graphical diagrams corresponding to different language objects. The graphical grammar is described using a form of BNF extended with operators denoting graphical relationships: ***contains***, ***is connected to***, etc.

[3] A channel is a communication entity which conveys signals between two designated agents. A detailed description of the communication facilities in SDL can be found later in this section.

[4] Also known as VDM-SL – Vienna Development Method Specification Language.

The *informal semantics* of SDL language objects is given in plain text in Z.100. Some language elements have stand-alone semantics, while others are only shorthand notation and their semantics is given by expansion into elementary language constructs. For example, an *output* statement which sends more than one signal is shorthand for a series of *outputs*, each one sending only one message. Thus, only the dynamic semantics of *outputs* sending one signal has to be defined. *Abstract syntax* also is not defined for shorthand constructs.

Finally, the *formal semantics* is given as a separate annex of Z.100 (Annex F, [IT99c]). It has two parts:

- A *static semantics* which provides well formedness rules for SDL models, written in first order predicate calculus, as well as transformation rules for shorthand notations.
- A *dynamic semantics* which provides an operational description of SDL model execution, in terms of Abstract State Machines (ASM, [Gur95, Gur97, Gur88]).

We discuss the semantics in more detail in §3.3. The formal semantics (Annex F, [IT99c]) has lower priority with respect to the rest of the standard. This means that whenever the informal semantics from Z.100 contradicts the Annex F, Z.100 takes precedence. The formal semantics constitutes one of the big differences between SDL-2000 and the previous versions of the language, in which both static and dynamic semantics were defined using Meta IV [ISO96].

3.2.2 Architecture and communication

Agents

As part of the functional description of a system, SDL supports the description of both structure and behavior. On the structural side, SDL has facilities for describing the architecture of a system in a hierarchical way, so that the complexity of a model can be managed one level of detail at a time.

Thus, the system components (called *agents*) form an *aggregation tree*, in which each agent (except the root agent which represents the entire *system*) is embedded in another agent from the upper level of detail. An agent encapsulates the contained agents and provides a black-box view for the outer agents. Communication is possible by means of asynchronous signals, either between sibling agents, or between an agent and its contained sub-agents. An agent may act as a router for the signals coming from or going to its sub-agents (this is usually the case for *block* agents, see below).

There are two kinds of agents:

- *concurrent* agents, in which sub-agents execute in parallel. This does not mean that in an implementation of the SDL system, these agents must be implemented in true parallelism or using the operating system multitasking. The *concurrency* attribute is a way of specifying that no constraint should be assumed about the possible interleaving of the actions of the contained sub-agents.

  For traditional reasons, concurrent agents are called *blocks*, with the exception of the top level agent which is always concurrent and is called *system*.
- *alternating* agents, in which sub-agents execute in a mutually exclusive way: when a sub-agent is executing a transition, every other sub-agent has to be in a stable state. The other sub-agents remain in the respective states until the executing sub-agent finishes the transition.

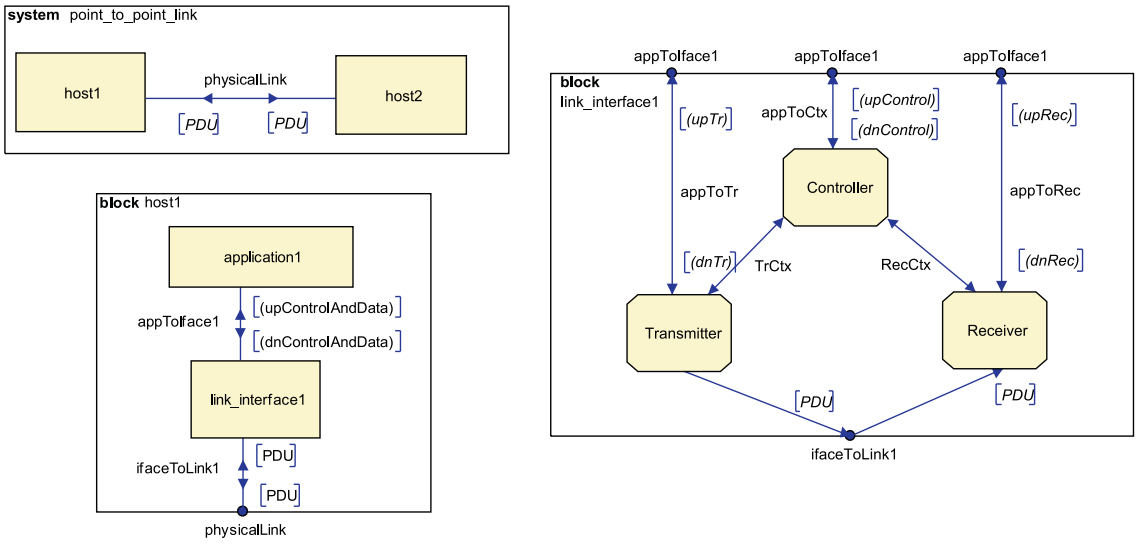


Figure 3.1: Hierarchical description of a point-to-point link in SDL

For traditional reasons alternating agents are called *processes*. Processes also support shared data, which may be used as a communication mechanism between alternating sub-agents inside a process, in addition to the asynchronous signal passing mechanism.

An example of hierarchical system architecture is presented in Fig. 3.1. It shows the decomposition of an SDL system modeling a point-to-point network link, into agents modeling the hosts, the link interfaces, their sub-components, etc. More concrete examples can be found in later chapters. In the figure, rectangles represent *blocks*, rounded rectangles represent *processes*, and lines between agents represent the *communication paths*, annotated with the types of signals they may carry. The graphical SDL symbols are shown in Fig. 3.2.

Each SDL agent can have its own behavior described through an extended state machine (this is discussed further on in §3.2.3). Each agent also has a unique identifier, called PId, which can be used by other agents to communicate with it by direct addressing (see next section).

There are several differences between SDL-2000 and SDL-96[5], with respect to architectural decomposition. The intention in SDL-2000 was to harmonize the two types of architecture objects (blocks and processes):

- In SDL-96, the behavior of blocks cannot be described by means of a state machine. Blocks do not have a PId and cannot themselves handle signals. They can only route signals (statically) towards inner or outer system components, by describing channel interconnections.
- In SDL-96, blocks cannot be created and destroyed dynamically, whereas in SDL-2000 they can.
- In SDL-96, blocks and processes cannot be mixed inside a block. In SDL-2000, they can be freely mixed.

[5]This stands also for previous versions of the language (SDL-88 and SDL-92). Henceforth, previous versions are mentioned only when they differ significantly from SDL-96.

– In SDL-96, processes cannot be refined in sub-processes. Instead, processes can be refined into entities called *services*, whose behavior is described with state machines. However, a SDL-96 service does not have its own identity (PId), and therefore signals cannot be sent to a specific service, but only to the whole enclosing process.

Communication methods

The primary communication method between SDL agents is by asynchronous *signals*. Signals are named entities which can carry data parameters. Signals are produced by an agent using an *output* instruction (discussed in §3.2.3). Upon arrival to a destination agent, a signal is placed in the input port of the agent. Each agent has an input port with a *signal queue*. Signals can be consumed, saved for further use or discarded by the agent state machine. More details are provided in §3.2.3.

Other communication means are:

– Remote procedure calls. An agent can call a *remote procedure* defined in another agent, if the communication paths between the two agents are properly specified. Remote procedure calls are actually realized by an implicit signal interchange, therefore they are only a shorthand notation.

– Remote variables. An agent may declare a variable as "remote", so that other agents may consult its value using an *import* instruction. Other agents actually consult *a copy* of the variable, which is updated explicitly by the exporting agent using an *export* instruction. Like remote procedure calls, importing remote variables is also realized by implicit signal interchange, so remote variables are also just shorthand notation.

– Variables shared by a *process* agent for the use of its sub-agents.

Signal-based communication can use either direct addressing or implicit signal routing. Direct addressing is done by specifying the destination agent's PId in the *output* instruction. A *route* (see next section) to the destination agent capable to transport the signal must nevertheless exist. Implicit routing is done when no destination PId is given. In this case, if routes to several destinations exist, one is chosen arbitrarily.

Channels and gates

Signals are conveyed through *channels*. A channel has two ends, each of which can be connected to an agent. The channel can be unidirectional or bidirectional. For each direction, the channel is considered to transport the signals reliably, i.e. without *loss*, *corruption* nor *reordering*. The channel may however delay the arrival of the signals, if a **nodelay** clause is *not* present in the channel definition.

Channels are connected to agents through *gates*. Conceptually, a gate is a couple of an input port and an output port. A gate can be connected (implicitly or explicitly) to channels both on the outer side of the agent, and on the inner side of the agent (if the agent contains sub-agents). In this case, the gate only transfers the signals from an outer channel to an inner channel, or vice-versa. Alternatively, a gate can be connected on the inner side directly to the agent state machine (connection can be implicit or explicit). In this case, the gate transfers signals from outer channels to the agent's *signal queue*, and signals produced by the agent (using *output* instructions) to the outer channels. Both gates and channels specify *statically* which signals can be transferred in either direction.

There are several differences between SDL-2000 and previous versions, concerning communication:

- In SDL-96 gates are used only in the definition of *agent types* (which are discussed further in this section). In SDL-2000, type-based agent definitions and non type-based agent definitions have been harmonized. As a consequence, gates can be defined in non type-based agent definitions.

  Additionally, implicit gates are sometimes created, for example when an agent defines a channel-to-channel connection directly without defining a gate.

- SDL-2000 introduces the concept of *interface*. An interface is a *named* collection of signal definitions, remote procedure definitions and remote variable definitions. Among others, interfaces can be referenced in gate definitions, when specifying the signal types transferred through the gate.

- Since SDL-96, channels are created implicitly in certain cases, e.g. when there is no explicit connection for a gate, and there is another gate in the same scope with a matching set of conveyed signals. This is done in order to avoid the overhead that channel definition sometimes causes for the modeler.

Type based modeling

In this paragraph we discuss type-based modeling of *agents*. However, all features described here are also available for *data types*, which are examined in §3.2.4.

SDL facilitates reuse by allowing type-based modeling of agents. Thus, if several agents with identical structure and behavior appear in different places in the system (e.g. *host1* and *host2* in Fig. 3.1), the behavior and structure of the agents can be described through a unique *agent type* which is afterwards referred from the respective places. This facility exists in the language beginning with SDL-92.

Type-based agents also facilitate reuse by including two mechanisms available in modern object-oriented languages: type *specialization* and *generic* types. With *specialization* (inheritance), a type can be derived from another by adding or modifying both the structure and the behavior properties. Specific restrictions to preserve "observational" type compatibility apply.

Generic data and agent types can be defined by using *context parameters*. Various language objects can be used as context parameters, including: agent types, procedures, variables, timers, gates, exceptions, etc. When a type with context parameters is instantiated, a concrete object of corresponding kind has to be provided for each formal context parameter of the agent type.

Agent types provide agent definition patterns, and their main use is to factor out the definition of identical agents appearing in several places in a system. However, agent types also relate to the data type system of SDL. An agent type A implicitly defines an interface type I_A (based on the signals, remote procedures, etc. that are handled by the agent type). In turn, each interface I defines a type T_I which is a sub-type of the predefined sort PId. A variable of type T_I contains a PId which points to an agent implementing the interface I.

By supporting data sub-typing, including for PId types, SDL supports *polymorphism*. Moreover, SDL provides type-safe polymorphism, as the data type system of SDL supports both static and dynamic typing, so for example the "real" type of a PId variable can be checked at run time. This is true for the entire type system of SDL and not only for the part referring to PId types.

PId sub-types and the dynamic typing system are new in SDL-2000.

3.2.3 Behavior

The previous section described the SDL constructs for architectural modeling and interfacing between system components. In order for the system to achieve the desired functionality, the behavior of each component (agent) has to be described. This section examines the computation model of SDL and the SDL constructs for describing behavior.

Control

An important aspect of a functional model such as SDL is the way threads of control are organized. Concurrent object oriented languages are usually classified in two categories with respect to ownership of control threads [Weg87, Pap92]:

1. *Orthogonal* languages, in which threads of control are independent from (orthogonal to) the object structure of the system. Usual sequential object oriented programming languages like C++ [Str97] or Smalltalk [GR89] have orthogonal models, in the sense that execution threads provided by the operating system can be freely used in programs.
2. Languages with *active objects*, in which threads of control are owned by certain objects (active objects). Furthermore, languages with active objects can be *homogeneous* – with only active objects, or *heterogeneous* – with both active and passive objects. Examples of such languages are POOL-T [Ame87] or Eiffel// [CR96].

From the point of view of this classification, SDL falls into the second category. In SDL, each agent has its own thread of control, which is created and destroyed together with the agent. The agent state machine specifies what is executed on that thread of control. The execution is marked by moments in which the thread is idle – when the agent is in a stable *state*, and by moments in which the thread is actually executing – when the agent executes *transition code*. States and transition code are described in the next paragraphs.

States

In SDL-96 state machines have a flat structure, meaning there is no hierarchical structuring of states. A *state* is just a *named* entity, used to partition the flow of control of an agent. States designate points in the flow of control where the agent stops and waits for a certain condition before continuing. The condition is usually triggered by the agent's environment (i.e. the other agents, the underlying machine or the system's environment):

- The arrival of a signal in the agent's input port, tested with an **input** clause, and possibly conditioned by a boolean test in a subsequent **provided** clause.
- The satisfaction of a logical condition, tested with a **provided** clause. The truth value of the condition may depend on the agent's environment, e.g. if the expression involves shared variables or the value of current time (**now**).

The agent may also resume execution automatically, without waiting for a change in the environment. This is done either by using a spontaneous transition clause (**input none**) or by testing a condition that holds without the intervention of the environment, such as **provided true**. The insertion of such states in which the execution is resumed automatically can be useful, for example in order to provide interruption points in a computation process.

In SDL-2000, several artifacts for state modeling have been added, although without changing the fundamental idea of state machines. The purpose of these artifacts is to facilitate the design of large state machines. They are:

- *Composite states.* A composite state groups a set of semantically related states of an agent. Composition can be used to factor out transition code: if a same transition can be triggered from several states, these can be grouped under a same composite state and the transition may be written only once.

  Composite state entry points and exit points, history nodes and other constructs make it easier to specify and understand the control flow in a behavior description. However, they do not add expressive power to SDL, in a strictly semantic sense.

- *State aggregation.* An aggregate state is a composite state with sub-states executing in parallel. The sub-states are also composite states, so each of them defines its own state-transition graph. They execute in parallel by interleaving, one entire transition at a time.

  Due to the interleaving semantics, an aggregate state can be transformed in a semantically equivalent flat composite state (by considering the cartesian product of the sets of sub-states). Therefore, state aggregation does not add expressive power to SDL, but only offers means to express more clearly the behavior of complex agents.

- *Entry and exit actions.* These actions, executed at the end of transitions entering a state or at the beginning of transitions exiting a state, are further means to factor out recurring behavior.

Although they do not add expressiveness to the language, the mechanisms presented above are valuable from a methodological point of view. They were initially introduced by D. Harel in the Statecharts formalism [Har87]. The ideas originating in Statecharts were later included in several object-oriented analysis and design methodologies [CD94, RBP+91], and in the Unified Modeling Language (UML, [OMG99]).

Transition code

An agent fulfills its functionality by executing actions during *transitions* from one stable state to another. From the SDL specification point of view, transitions are not always clearly identified entities originating in a stable state and terminating in another. This is because control flow structuring constructs such as *branching* and *jumps* can be used in transition specification. For this reason, we prefer the terminology *transition code*[6].

The transition code of an agent is structured on a state/clause basis: a code sequence is attached to a certain stable state and to a certain clause. Two types of clauses (**input** and **provided**) have been mentioned in the previous section. Besides them, two additional clauses are defined in SDL:

- **priority input**, which has the same meaning as **input**, except that the specified signal does not have to be first in the agent's signal queue (it is consumed regardless of its position in the queue).

[6] For language definition simplicity, in [IT99b] transitions are considered separate entities. Branching constructs and join points introduce pseudo-states from which transitions can originate or in which they can end.

– **save** specifies that a signal is not consumed by the agent in the current state, but it is saved in the queue for further use. In the absence of a **save**, if a signal is in the head of an agent queue and there is no corresponding **input** clause in the current agent state, the signal is discarded.

  A **save** clause may not be followed by further transition code.

The graphic symbols representing clauses are shown in Fig. 3.2.

The code sequence following an **input**, **priority input** or **provided** clause can contain basic statements (which have a graphical representation and existed before SDL-2000) and compound statements written in the textual algorithmic language that was added in SDL-2000. The basic statements are (see Fig. 3.2 for the graphical counterpart):

– *Informally specified actions* (**task**). These are actions specified informally with a string containing plain text. Formally, they have no effect and are just a placeholder to be used during analysis/design.

– *Assignments and assignment attempts* (**task**). Used to assign the result of an expression to a variable, parameter, etc. Assignment attempts are specific to SDL-2000 and perform a dynamic type checking before assignment.

– *Agent creation* (**create**). Used to create a new agent. In SDL-96, the agent can only be a *process*, and the create instruction must specify the *process instance set* in which the new agent is created. The process instance set specifies the channel connections of the newly created instance (shared with already existing instances in the set).

  In SDL-2000, both blocks and processes can be created dynamically. The **create** statement may either specify an *agent instance set*, or just an *agent type*. In the latter case, the agent instance set in which the agent is created may be chosen from the existing sets of the same type, or a new set with implicit connections may be created if no sets of the same type exist.

  The execution of a **create** statement updates several implicit variables. In the initial agent, the **offspring** variable holds the PId of the newly created agent. In the newly created agent, **self** holds its own PId, while **parent** holds the PId of the agent executing the **create**.

– *Signal output* (**output**). The statement creates a new signal instance, with parameter values specified in the **output** statement. The signal destination may be specified by *direct addressing* (using the destination agent's PId) or *indirectly* (using an output gate of the agent, and relying on the default routing mechanism of SDL– this may imply non-deterministic choices at certain points).

  In both cases, the signal is conveyed by a channel route, determined at the moment the **output** is executed, and the delays and queuing order of the route apply.

  The PId of the sender process is sent with the signal. When the signal is consumed by the destination agent using an **input** clause, this PId is stored in the implicit variable **sender** of the destination agent.

– *Procedure calls* (**call**). This statement can be used either to call a procedure on the control thread of the current agent, or to send a *remote procedure call*.

  In the latter case, an implicit signal representing the procedure call is sent to the remote agent. The same conditions from the above description of **output** apply in this case.

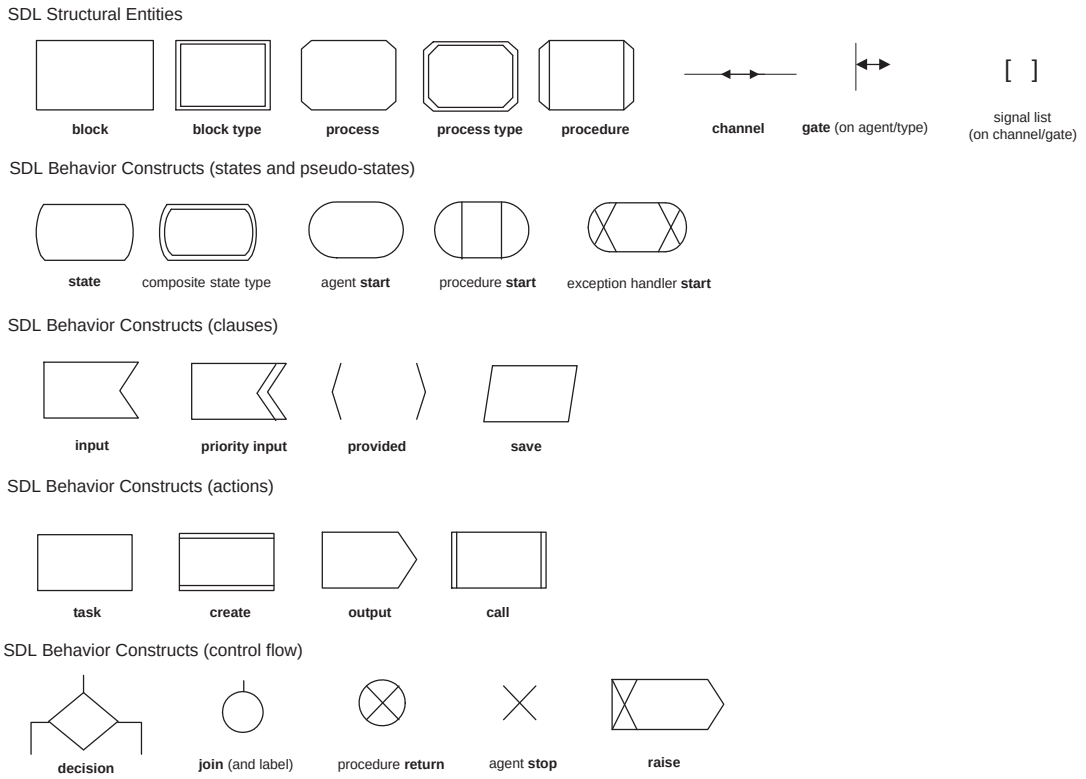


Figure 3.2: Graphical SDL symbols

Additionally, a watchdog timer may be set in parallel with the procedure call, to unlock the caller agent in case of non-response from the remote agent.

– *Timer management* (**set** and **reset**). Timers are discussed in a dedicated subsequent paragraph.

Additionally, SDL contains the following graphical control flow structuring statements:

– *Decisions* (**decision**) are used for conditional branching. The condition may be formal or informal (written in plain text). If the decision discriminant is the keyword **any**, the decision equates a non-deterministic choice between the specified decision *answers*.

– *Jumps* (**join**) are used for unconditional branching. Any basic statement may be preceded by a *label*, which can be used in **join** statements.

– *Return from procedure* (**return**) can be used both in local and remote procedures, and it can specify a return value.

– *Termination of agent execution* (**stop**), stops the execution of the agent executing the statement.

– *Raising a software exception* (**raise**). The exception mechanism, specific to SDL-2000, is examined in a dedicated further paragraph.

Textual algorithmic language

As mentioned before, a textual algorithmic language has been added in SDL-2000. We do not aim to describe this language here, but rather to give a general idea of the supported features. The statements of this language resemble (in both a syntactic and semantic way) the statements of the C++ programming language [Str97]. They are:

- *Compound statements*, with the possibility of defining variables local to the compound statement.
- *Expression statements*. As in C and C++, the evaluation of an expression can constitute a statement.
- *Conditional statements*, which provide a simpler alternative to decision statements. Textual *decision statements* are also supported.
- *Loop statements*, with flow control similar to C and C++, including **break** and **continue** statements.
- *Exception handling*, with a construct similar to the C++ try-catch statement (**try-handle**, in SDL-2000).

Additionally to the above mentioned statements, all the transition actions with graphical representation enumerated in the previous paragraph can be included (in textual form) in a textual algorithmic language statement.

Compound statements of the algorithmic language can be placed inside **task** instructions on transition code. They can also be used for describing the body of SDL procedures.

Exception handling

A major addition to SDL-2000 is the exception mechanism. Exception handling is a useful programming technique available in many programming languages including Ada [Eng96], C++ [Str97], Eiffel [Mey95] and others. It allows the programmer to tackle in an organized way with exceptional situations that may appear in a software system due to either hardware or software malfunction or mishandling. For a more comprehensive discussion on the topic of exception handling in programming languages in general, the reader is referred to [Set96] and [Mey97].

In SDL, there is a set of predefined exceptions which are raised by the underlying abstract SDL machine. The modeler may also define his own exception types; such software exceptions can be raised using a **raise** statement.

Once raised, an exception propagates up on the procedure call stack of an agent. Exceptions raised in remote procedures also propagate back to the caller agent. If an exception reaches the agent level and is not handled, the further execution of the SDL system is undefined.

Exception handling is done by attaching handlers to different SDL objects. An exception handler is a *named* entity containing a set of exception handle clauses. A handle clause resembles a normal state machine **input** clause: it specifies the type of exception (signal) being handled and is followed by transition code.

A handler is defined within the scope of an agent or a procedure. However, the code portions on which a handler *is active* may be finer grained: the handler may be attached to the entire state/transition graph of an agent/procedure, to a composite state, to a simple state and its transitions, to a transition, to another handler, or to just a single action.

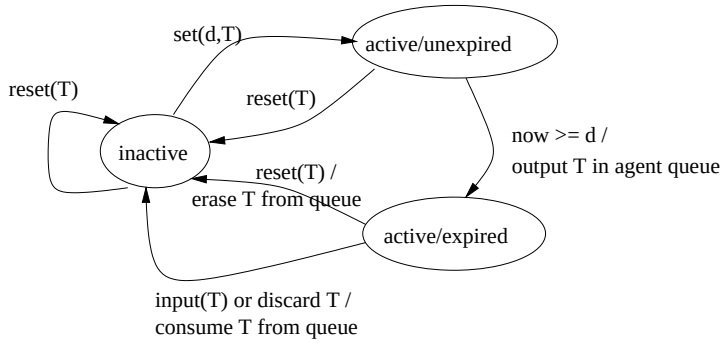


Figure 3.3: The behavior of a timer T. d is a Time value for the deadline.

Describing time-related behavior

SDL provides facilities for describing time-driven behavior. This is an important feature of the language, as SDL targets the specification and design of real-time systems. As we will see in a later section (§3.5), the facilities for describing timing information in SDL address the design or implementation level; however, they have a limited usability in the initial phases of system specification, when more abstract and descriptive timing information needs to be captured.

SDL provides two predefined data types related to time, Time and Duration, and predefined operators for handling these types (adding Time with Duration, multiplying Duration, etc.) The current time – i.e. the time since the beginning of system execution – may be consulted using a predefined operator, **now**. The manner in which time progresses during the execution of a system is, however, left unspecified in SDL. The only assumption that can be made is that successive evaluations of **now** yield (non-strictly) increasing results.

Time-driven behavior may be described either by using directly the value of **now** in tests or transition triggers, or by using *timers*.

Timers are special objects of the SDL language. They resemble data objects from certain points of view, but also have their (predefined) behavior which parallels the behavior of the system agents. Each timer definition also introduces a new implicit signal type, with the same name and parameter types as the timer.

A timer can be declared by an agent, using the **timer** keyword. Optionally, the definition may contain a default relative deadline for the timer. The behavior of a timer, as described by the SDL standard, is sketched using a simple state machine in Figure 3.3. There are three predefined operators on timers:

- **set**(*Time-value*, *timer*), which arms a timer with the deadline specified by the Time value. If time elapses beyond that deadline, and the timer is not reset in the meantime by the agent, the underlying abstract SDL machine switches the timer to the *expired* state, and puts a signal corresponding to the timer in the agent's signal queue.
- **reset**(*timer*), which switches the timer back in an *inactive* state. If the timer has expired beforehand, the signal corresponding to the timer is erased from the agent queue.

  A signal returns in the *inactive* state either when it is **reset** or when the corresponding signal is consumed from the queue by an **input** or a **discard** clause.
- **active**(*timer*), is a query operator which returns the boolean value *true* if the timer is in one of the two *active* states shown in Fig. 3.3, and *false* otherwise.

It is important to note that timer expiration signals always pass through the signal queue. Therefore, when a timer is consumed with an **input** clause, the only assumption that is guaranteed by the semantics of SDL is that the deadline of the timer has passed. In principle, nothing can be assumed about how much time has passed since the timer has expired. This is a comfortable semantics from the implementation point of view. However, it has certain disadvantages when SDL is used for building initial abstract models of a system, as discussed further on in §3.5.

3.2.4 Data

SDL provides a data system similar to that of usual imperative programming languages. The data system is an important part of any modeling language, since even the most simple systems involve the manipulation of a certain amount of data. However, for the purpose of this thesis, which focuses on the specification and validation of timing properties in real-time systems, data in SDL is not of paramount importance. Therefore, in this section we only give a brief introduction to the SDL data system.

Data types

SDL provides predefined types, such as Boolean, Character, Integer, Real, Time, Duration, etc., and mechanisms to define more complex types based on simple ones. The mechanisms for defining new types include constructs similar to what can be found in Algol-like programming language, as well as constructs inspired from object-oriented languages.

In the first category, we mention constructs for creating record types (**struct,**) records with variants (**choice**), enumerated types (using **literals**), sub-range types (**syntype**), or *collections* of different elements types (based on several predefined collection type generators: Array, String, Bag, Powerset, Vector). As a general note, the set of type definition constructs suported by SDL is larger than that of most usual programming languages.

Object-oriented concepts are included in the SDL data system: a data type definition may include *operators* and *methods* which act over values of that type, and inheritance relationships between data types may be defined. Inheritance allows the redefinition of operators and methods, and additions to the type structure, as in usual object-oriented languages.

As mentioned before, the data type system of SDL supports dynamic type checking and includes PId types which provide typed references to *agents*.

PId sub-types, dynamic typing, as well as several other object-oriented features are newly introduced in SDL-2000.

Variables and parameters

Data types are used in order to define *variables* and *parameters* at different levels in an SDL description.

The notion of *variable* in SDL corresponds to the same notion from imperative programming languages: variables have a name and may store a value of a certain type. Variables may be defined in different entities: in an agent, a procedure, a composite state or a composite algorithmic action. The *lifetime* of the variable is equal to the lifetime (activation time) of its enclosing entity. The *scope* of the variable is the enclosing entity, and in some cases its sub-entities. For example, a variable defined in a *process* agent is visible in all its sub-agents and

in the procedures defined therein. A variable defined in an agent or procedure is visible in all procedures recursively defined in that agent or procedure.

For complex types such as object types, both *expanded* variables – which contain an object with its fields, and *reference* variables – which contain only a reference to an actual object, may be defined. This facility is new in SDL-2000.

Parameters designate named data items, and are used for passing around data values in agent creation, procedure calls or operator/method calls. The semantics of parameters is similar to that from procedural languages.

Data types may also be used in the definition of *signals*, *exceptions* and *timers*. They designate the types of data items that may be conveyed by these communication objects.

Expressions

Expressions are SDL syntactic constructs for obtaining data values. An expression may involve:

- literals designating predefined type values (e.g. `1`, `true`, `0.5`) or values of structured types (e.g. `(. true,1 .)` – designating a structure with two components, a boolean and an integer)
- variable or parameter identifiers
- calls to user defined operators, value-returning procedures and methods
- predefined operators, etc.

3.3 Semantics

SDL is an object-oriented modeling language used in the development of real-life applications. One of the features which differentiate it from other languages from this category is the definition of its semantics.

While other modeling languages, such as OMT [RBP+91], ROOM [SGW94] or UML [OMG99, RJB98, BRJ98], only provide an informal semantics for the language concepts, the definition of SDL [IT99b] contains a formal semantics for the entire language [IT99c], i.e. a way of mapping any SDL specification to a clearly defined mathematical object. Thus, SDL is part of a family of standard languages with formal semantics, generically known as Formal Description Techniques (FDT's), family which also includes LOTOS [ISO89b] and ESTELLE [ISO89a].

The semantics of SDL fulfills two functions:

- defines formally the notion of *well-formed* SDL *system*, and
- provides a mathematical interpretation for the notion of SDL *system execution*.

The two parts are relatively independent, and constitute respectively the *static semantics* and the *dynamic semantics* of SDL.

While a formal *static* semantics for SDL consists only in representing the well-formedness conditions from the informal language definition [IT99b] in a formal language such as first order predicate calculus, the definition of a *dynamic* semantics implies a choice of a base formalism which is less obvious and has more implications as to the analysis methods applicable to SDL specifications. For this reason, several dynamic semantics for SDL have been proposed in the literature.

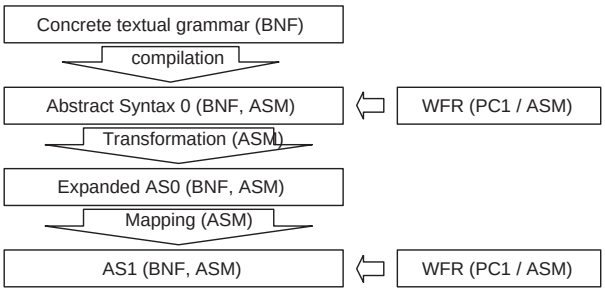


Figure 3.4: SDL static semantics layers.

In the 1988 version of SDL, the ITU–T standardized a semantics based on the Meta-IV language [ISO96]. This semantics was further updated with the release of SDL-92. As SDL-2000 brings along many changes, the ITU–T preferred to redefine the semantics from scratch rather than update the existing semantics. Another formalism, Abstract State Machines (ASM, [Gur88, Gur95, Gur97]) was preferred to Meta-IV for this task. In the following sections, we base the discussion on the new semantics of SDL-2000, and we briefly mention the other proposals found in the literature.

3.3.1 Static semantics

The goals of the static semantics are:

- to define the notion of *sound* SDL system, and
- to provide a basis for the definition of the dynamic semantics.

A first level of soundness, as in any language, is defined by the syntax. Z.100 [IT99b] contains a concrete textual syntax described in usual Backus-Naur Form (BNF). There are however additional constraints that a correct SDL system must observe, which cannot be expressed directly through a context-free syntax. Static typing constraints or identifier scope rules are examples of such constraints. These rules are defined in the static semantics.

Moreover, the concrete syntax of a complex language such as SDL is difficult to use directly as a basis for the definition of the dynamic semantics. For this reason, the static semantics is organized as a set of increasingly abstract layers, shown in Fig. 3.4.

In the less abstract layer (topmost), an SDL specification is modeled by its syntactic form. A syntax tree can be built from the model through a compilation process. By removing the unnecessary tokens from this tree (separators, various terminals), a more abstract tree containing only the meaningful language objects is obtained. This abstract model of an SDL system corresponds to the *abstract syntax level 0* (AS0) of SDL, defined by the static semantics.

Two more layers are added in order to ease the definition of the dynamic semantics. This is done by identifying a set of basic mechanisms, for which a dynamic semantics is defined, and translate other language constructs in terms of the basic mechanisms. For example, signal exchange is a base mechanism in SDL, whereas *remote procedure calls* are actually realized by an implicit *signal* exchange. Only the basic constructs are included in the layer 1 of the Abstract Syntax (AS1), which corresponds to the bottom layer in Fig. 3.4. The static semantics gives the rules for *transforming* an AS0 tree with *remote procedure calls* in an AS0 tree using only

outputs. The AS0 tree obtained after transformation, called *expanded AS0* tree in Fig. 3.4, is then mapped into an AS1 tree.

The structure of AS0 and AS1 trees is given in [IT99c] in a formalism similar to BNF. However, the non-terminals and productions of AS0 and AS1 also define ASM domains, and respectively access functions corresponding to each kind of language object (see the introduction to ASM in the next section). The *well-formedness rules* for SDL systems are then given as formulae of the first order predicate calculus (PC1) over these functions/domains.

3.3.2 Abstract State Machines

The base formalism used for defining the dynamic semantics of SDL in Annex F of Z.100 [IT99c] are the Abstract State Machines (ASM). This section is not intended as a full tutorial for ASM, and provides only the definitions needed to understand the rest of the thesis. For a thorough introduction to ASM, the reader is referred to [Gur95, Gur97].

For each sound SDL system, [IT99c] defines a corresponding *multi-agent ASM*. For simplicity, we define first the notion of *mono-agent* ASM.

Definition 3.1 (Abstract State Machine) An ASM is a tuple $A = (V, S_0, P)$ with the following components:

1. V denotes a *vocabulary* (or *signature*) of domain names, function names and predicate names.
2. S_0 denotes an initial *state* (or *interpretation)* of the vocabulary V.
3. P is a *program* iteratively modifying the interpretation of the vocabulary V.

The meaning of the components described above is detailed in the following. Besides being a model, ASM is also an algebraic specification formalism, with its own established syntax. The syntax is presented in parallel with the model elements below.

Vocabulary

The *vocabulary* V contains domain names, function names and predicate names. Each function or predicate has an *arity*, which specifies the domain names of the parameters and of the result. Predicates always have the result in the *Boolean* domain.

Boolean is a special domain name defined by any ASM. Several other domains (*Nat*, *Real*) and usual functions (arithmetic and boolean operation names) are also considered to be defined by any ASM. Moreover, the interpretation of these "predefined" domains and functions is considered to correspond to their standard mathematical definition.

The vocabulary of an ASM is given by declaring the domain names and function names and arity, using the following syntax ([attributes] denote optional properties such as **static**, **controlled**, **monitored**, **shared**, described in the following paragraphs):

[attributes] **domain** D

[attributes] $f : D_1 \times D_2 \times ... \times D_n \rightarrow D$

States. Initial state.

A *state* is a function that assigns a mathematical *interpretation* to each domain name and function name in the vocabulary. The interpretation of a domain name D in a state s must be a set, denoted $s(D)$. For a function name f with the arity $f : D_1 \times D_2 \times ... \times D_n \to D$, the interpretation is a function $s(f) : s(D_1) \times s(D_2) \times ... \times s(D_n) \to s(D)$.

For simplicity, all the domain name interpretations (called domains henceforth) are considered to be part of an infinite set $s(X)$, which is the interpretation of a predefined domain name X called the *base set* of the ASM. $s(X)$ contains a particular element called *undefined*, and there is a predefined function name $undefined :\to X$ which denotes this element. All functions defined by an ASM are considered *total* on X, and they yield *undefined* for all elements for which an interpretation is not explicitly defined.

In the specification of an ASM, the initial state S_0 of an ASM is given by a set of **initially** clauses, as shown in the example below:

initially $D = \{el1, el2\}$

initially $\forall d \in D : f(d) = 1$

Static and dynamic names

As we mentioned before, the interpretation of some elements of the vocabulary V is fixed a priori. For example, the interpretation of the *Boolean* domain name must always be a set of two elements representing the values *true* and *false*, and the predefined boolean operator names must have an interpretation corresponding to their mathematical definition.

Such names are called **static** names. Their interpretation is either fixed by the initial state of the ASM (S_0) or predefined by the ASM framework and must not change during the execution of the ASM (the notion of execution is defined below).

The names whose interpretation may change from one state to another are called **dynamic**. Both functions and domains can be static or dynamic.

Basic and derived names

Both domains and functions may be *basic* or *derived*. *Basic* names have their own interpretation, which is either predefined, or defined by the initial state S_0 and modified by the ASM program. *Derived* names have their interpretation derived from the interpretation of the basic names.

For example, a derived domain can be defined as:

$$D =_{def} D_1 \times D_2$$

and a derived function can be defined as:

$$f(d : D) : D' =_{def} g(\textbf{s-}D_1(d), \textbf{s-}D_2(d))$$

where g is a function defined on $D_1 \times D_2$.

In the above examples, we have used some function names that are implicitly defined for derived domains. For example, for the product domain D above, the functions $\textbf{s-}D_1 : D \to D_1$ and $\textbf{s-}D_2 : D \to D_2$ are implicitly defined and can be used to extract the components of a couple $d \in D$. The constructor function $\textbf{mk-}D : D_1 \times D_2 \to D$ builds a couple from the components.

Programs and runs

The program P of an ASM $A = (V, S_0, P)$ specifies how the state of the ASM is updated. There are two kinds of elementary constructs of an ASM program:

- the *location update rules*, which have the following form:

  $$f(t_1, ..., t_n) := t_0$$

  where f denotes a *basic dynamic* function and t_0,t_1,...,t_n denote *terms*[7] constructed from ASM function names.

  To execute the above update in a state s means to transform s into a state s' such that $s'(f)(s(t_1), ..., s(t_n)) = s(t_0)$, and the interpretation of all the other elements of V remains the same in s' as in s.

- the *domain update rules*, which have the form:

  extend D **with** d
  ... // further rules
  endextend

  where D denotes a dynamic (basic) domain. To fire the above update in a state s means to transform s in a state s' in which the domain $s'(D)$ has an additional element compared to $s(D)$. The new element is denoted by the term d in the update rules enclosed by **extend**..**endextend**, so specific location updates referring to this element may be written.

More complex transition rules may be constructed recursively, based on simple location and domain updates: *conditional* updates (**if** *condition* **then** *Rule1* **else** *Rule2* **endif**), *parallel* updates *(Rule1 Rule2 ...)*, *non-deterministic* updates (**choose** v : *condition*(v) *Rule*(v) **endchoose**). However, for a given state s, a program P always resolves to a set of elementary updates, which are executed in parallel. A soundness requirement for the ASM is that parallel updates are not contradictory, i.e. for each distinct location (basic dynamic function + parameter values) there is only one update.

The executions (*runs*) of a single agent ASM are modeled through finite or infinite sequences of state transitions of the form: $s_0 \xrightarrow{P} s_1 \xrightarrow{P} s_2 \xrightarrow{P} ...$, where $s \xrightarrow{P} s'$ denotes the application of the updates specified by the program P in the state s to obtain the state s'.

As it can be noticed, the execution of an ASM means iteratively applying the *same* program P to the current state of the ASM, over and over again. However, as *conditional* updates are possible, the same program P may specify different update sets in different states of the ASM.

Multi-agent ASM

Mono-agent ASMs define execution as a sequence of applications of the ASM's program to the ASM's current state, resulting in a sequence of states. The updates are executed by an implicit agent, which is not described explicitly in the ASM model.

In multi-agent ASMs (also called *distributed ASMs*), there can be several execution *agents* firing state transitions simultaneously, and sharing the same ASM state. Moreover, the set of

[7] *Terms*, which are function applications, can be written either in the prefix notation shown above, in infix notation (for some predefined operators), or in an alternative dotted notation, as in: $t_1.f(t_2, ..., t_n)$. The parentheses may be dropped for nullary functions in prefix notation and for unary functions in dotted notation.

agents is dynamic, and the assignment of programs to agents is also dynamic. A distributed ASM contains several domain and function names with predefined meaning:

controlled domain *Agent*	-	contains one element for each concurrently executing agent
static domain *Program*	-	contains one identifying element for each program description
controlled *program* : *Agent* $\rightarrow$ *Program*	-	identifies the program executed by each agent
monitored *Self* : $\rightarrow$ *Agent*	-	function interpreted differently by each agent, provides the agent with its own identity

The concurrent execution that takes place inside a multi-agent ASM is modeled as a set of *partially ordered runs*. We do not intend to provide here a thorough introduction to the execution model of distributed ASMs. The model is detailed in [Gur95, Gur97], and several properties of partially ordered runs are deduced therein. We only note that some of the properties stated in [Gur95, Gur97] imply that *the partial order model is equivalent to an interleaving model* for the execution of the parallel agents, in the sense that the same ASM states are reachable and the same properties are held by the two models.

Distributed ASMs are used for the definition of the SDL semantics, as they provide a convenient way to cope with the inherent concurrency of SDL models. As we will show in §3.3.3, SDL *agents* are modeled through ASM agents, thus taking the burden of modeling concurrency out from the SDL semantics description.

Open specifications and real-time behavior

SDL systems are open, in the sense they can interact with an unspecified environment. ASM specifications can model the intervention of an unspecified environment through **monitored** or **shared** functions and domains (functions and domains which can only be updated by the ASM agents, introduced previously, are called **controlled**).

A **monitored** function or domain is a dynamic object that can only be modified by the *environment*. Thus, a **monitored** object can change its interpretation from state to state in an unpredictable way (unless prevented by some integrity constraints). An ASM agent may test the value of a monitored object, and take actions corresponding to the state of the environment modeled by that value. However, an agent may not modify (update) the values of monitored objects.

Shared functions and domains differ from **monitored** in that they can be updated both by the environment and by the ASM agents.

Integrity constraints (written as predicate calculus formulae in the ASM specification) may restrain the possible interventions of the environment on the **shared** and **monitored** functions.

An important case of interaction with the environment is represented by the elapse of time. In the semantics of ASM, time is considered part of the environment and can only be consulted but not controlled by the agents. This is modeled through a **monitored** function:

monitored *currentTime* : $\rightarrow$ *Real*

The integrity constraints imposed to *currentTime* are too complex to be expressed in predicate calculus over the usual terms admitted in ASM specifications. These constraints are an important part of the execution model of SDL. They basically impose the following conditions:

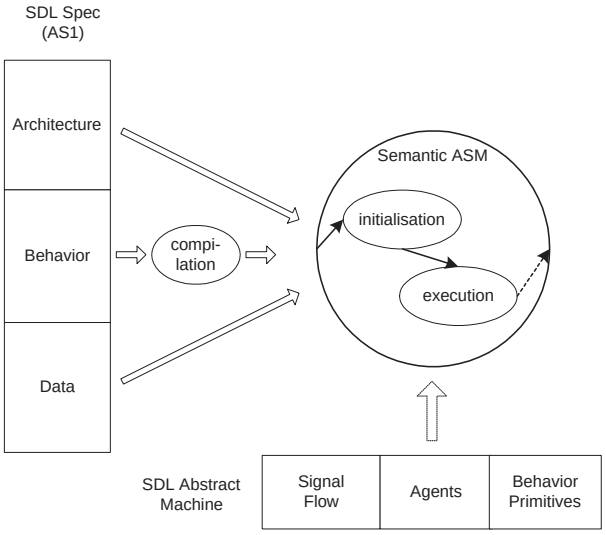


Figure 3.5: SDL dynamic semantics

1. *Monotonicity*: the *currentTime* function changes its values monotonically increasing over ASM runs.
2. *Discreteness*: for every $\tau \in \mathbb{R}$ there is a finite number of steps made by the ASM before *currentTime* becomes greater than τ.

3.3.3 Dynamic semantics

In this section we will examine the organization of the dynamic SDL semantics as described in [IT99c] and we will discuss the semantics of concurrency and time. We conclude with some general remarks on the suitability of ASM for describing the semantics of SDL, and on the other semantic definition attempts that can be found in the literature.

Structure of the dynamic semantics

The approach undertaken in [IT99c] to define the dynamic semantics of SDL is sketched in Fig. 3.5. The semantics provides the rules for building the ASM representing the semantic model of an SDL system.

The *semantic ASM* is a multi-agent ASM (see previous section) comprising one ASM agent for each living SDL agent, and one ASM agent for each agent instance set contained in the system, at a certain moment. The behavior of each agent, prescribed by a corresponding ASM *program*, comports two phases: an *initialization* phase and an *execution* phase. The entire semantic ASM of an SDL system is executed in two phases, which correspond to the initialization and execution of the *system* agent.

The programs of the semantic ASM are based on a set of ASM rules implementing the behavior of the basic SDL objects. These rules form together a library of ASM macros[8], called the SDL Abstract Machine (SAM). The description of the SAM takes up a significant part of the formal dynamic semantics of SDL, and includes primitives which implement:

[8] An ASM *macro* is a named *update rule*, that can be referenced from other macros or programs.

- *The signal flow model* of SDL: signals, gates, input ports, channels and routing, timers and exceptions.
- *The agent model* described above, comprising the definitions of agent domains and associated functions (agent mode functions for modeling phases of the agent's behavior, etc.).
- *The behavior primitives*, the abstract machine instructions of the SAM. This part contains rules implementing SDL statements: assignment, call, output, create, timer set/reset, etc.

The SAM programs which specify the behavior of the semantic ASM use, directly or through the SAM primitives, the syntactic structure of the SDL system, more precisely the AS1 syntactic tree extracted from an SDL specification (see the static semantics, §3.3.1). As shown in Fig. 3.5, the SDL structure and data parts are used directly, while the AS1 syntactic tree nodes corresponding to the behavior description, i.e. SDL state machine transitions, need an additional preprocessing step.

The preprocessing step, called *compilation* in [IT99c], is necessary because of the limited capability of ASM to represent *sequential* behavior. As shown in §3.3.2, an ASM agent functions by repeatedly evaluating the same program over the current ASM state, and atomically updating a set of locations as a result of this evaluation. In order to model sequential algorithms in ASM, one has to explicitly store and use control flow information, e.g. by keeping an ASM function that memorizes the current position in the program (*program counter*). This is how SDL transition code is handled in [IT99c]. The *compilation* step assigns unique labels to the SDL instructions contained on every transition in the system, used as values of the program counter.

A complete description of the components of the dynamic semantics of SDL described above is outside the scope of this work. For that, the reader is referred to [IT99c]. Some features of the dynamic semantics which are relevant to the present work are discussed in the next paragraph.

Semantics of concurrency

The concurrency model for SDL is derived from the concurrency model of distributed ASMs, described in 3.3.2. As we mentioned there, the runs of a multi-agent ASM are partially ordered sets of transitions, but the model is proved to be equivalent to a model with *nondeterministic interleaving* at the level of ASM agent transitions.

On the level of SDL, this model corresponds to *interleaving of individual* SDL *actions* (**task**, **create**, **output**, etc.), as all simple SDL actions are executed in one ASM step[9].

The evaluation of the expressions contained in an SDL action, e.g. **output** or **create** parameters, right-hand side of **task**, in not included in the unique ASM step mentioned before. These expressions are evaluated in a series of ASM steps preceding the action. Therefore, interleaving may occur during the parallel evaluation of expression in parallel agents. However, as there is no communication between agents executing in parallel[10], other than by means of SDL signals, and signals cannot influence the result of an ongoing expression evaluation, the interleaving that may occur during parallel expression evaluation has no influence over the overall behavior of the system.

[9] This concerns only simple actions (see §3.2.3). Compound actions written in the textual algorithmic language, are mapped to structures of simple actions in the static semantics section of [IT99c]. For compound actions, the atomicity level is that of the simple actions contained therein.

[10] Alternating agents contained in a *process* agent (see §3.2.2) are protected from parallel execution using a mutual exclusion flag in the state of the owner agent.

Semantics of time. Timers

As shown in §3.3.2 on page 53, time in the ASM model is considered part of the environment, and modeled through the **monitored** function *currentTime*. There are two monitored functions relevant for the temporal behavior of SDL systems in the dynamic semantics:

1. **monitored** $now : \rightarrow Real$, is used instead of *currentTime* in the SDL semantics, to represent the current time. *now* satisfies the same constraints as *currentTime* (monotonicity, discreteness), and one additional constraint:
   - The value of *now* does not increase as long as a signal is in transit on a non-delaying channel.
2. **monitored** $delay: Link \rightarrow Duration$ gives the amount of time with which a signal passing through a *Link* is delayed. *delay* is a monitored function, so it can vary nondeterministically during the execution of the system. It satisfies two integrity constraints:
   - It always returns 0 for non-delaying channels.
   - For every link l, successive evaluations of $now + l.delay$ yield increasing values. This constraint ensures that channels preserve the order of the conveyed signals.

Timers are managed in the SDL Abstract Machine using the concept of *schedule*. Each *input* gate has a *schedule* which contains a list of signals with their corresponding arrival times. In the ASM model, the *schedule* contains both the signals in transit (with the arrival time $> now$) and the arrived signals (with arrival time $\leq now$). The queue of the gate, which in SDL is a "physical" object, is merely a *derived function* in the ASM model: it contains the signals from the *schedule* for which the arrival time is $\leq now$. For further details on the modeling of *schedules* the reader is referred to [IT99c, GGP99].

The schedule provides a convenient way to handle timers: when an agent sets a timer, a corresponding timer signal is directly put in the agent's schedule, with an arrival time equal to the timer deadline. Then, as soon as *now* becomes greater than the deadline, the timer becomes visible in the (derived) queue of the agent.

Schedules are also used, in combination with the *delay* function, to model communication channel delays.

We outline below, in less formal terms, the main characteristics of the semantics of time in SDL implied by the ASM modeling described before:

- Individual actions on SDL transitions are atomic, and execute in 0 time.
- The evaluation of SDL expressions is not atomic, and therefore the value of the ASM function *now* may vary during evaluation. This may influence the result of expressions involving the SDL predefined expression **now**.
- Any amount of time may generally pass between the execution of two actions, or before a transition is triggered.
- A timer, although visible in the queue from the moment it expires (see the explanation on *schedules*, above) may be ignored by the concerned agent for an indeterminate amount of time.

ASM vs. other semantic approaches

It can be argued that the ASM semantics of SDL presented in the previous paragraphs, while being formal, captures the functioning of SDL systems at the right abstraction level. Indeed, the functioning of one ASM agent is close to that of an Extended Finite State Machine, which is the intended behavioral model of SDL agents [OFMP+94].

Moreover, *composition* and *communication* between parallel agents is captured in ASM by the multi-agent ASM model and by sharing parts of the state between multiple agents. This model corresponds naturally to the semantics of SDL described informally in Z.100 [IT99b]. The model also makes unnecessary the description of explicit *composition operators*, which would be necessary in an automata-based semantics model, while still being equivalent to an asynchronous-synchronized composition (the synchronized actions corresponding to modifications of the shared locations of the global ASM state).

The ASM semantics of SDL also responds to a number of critiques which concerned the previous version of the standard formal semantics (see for example [Boz99]), namely the concurrency model, the handling of timers, etc.

Several other semantic models for SDL have been proposed in the literature. They usually tackle only with a subset of the language. We mention some of them here, with an accent on those concerned with the representation of timing issues.

- [KM95, MGHS96] describes two possible ways of defining a timed denotational semantics for SDL, based on the Duration Calculus [CHR92]. The semantics of an SDL process is a duration calculus formula satisfied by the process specification, and the semantics of a system is obtained by the conjunction of the formulae corresponding to the system components.

  System timing hypotheses, e.g. duration of individual tasks, may be expressed similarly with duration calculus formulae. The semantics together with the hypotheses may be used for proving timing properties of the SDL system. However, we found no characterization of the level of automation of this task in the literature.

  The semantics given in [KM95] is restricted to a small subset of SDL (restrictions concern architecture, behavior and data) and cannot scale up without difficulties, as noted by its authors.

- A different approach is proposed in [BFG+99, BGMS98, Boz99], where a semantics is given to a representative subset of SDL by translation to another formalism, IF, based on extended communicating timed automata. The dynamic semantics of IF is described formally in [Boz99], using a layered approach and taking as basis Labeled Transition Systems (LTS) and Timed Automata (TA, [ACD93, AD94]). The translation of SDL to IF is described informally in [Boz99].

  The approach undertaken in [BFG+99, BGMS98, Boz99] answers many problems raised in the context of our present work, concerning the description and analysis of time-related behavior. However, the answers are given at the level of IF rather than SDL. In contrast, the approach presented in this thesis concentrates on providing extensions, semantics and techniques working directly on the level of SDL and its standard ASM semantics.

- Other semantic models for SDL proposed in the literature are not particularly concerned with timing issues. We cite here semantic definitions for SDL based on Petri Nets [FG97, FDT95], process algebra [BMU98], finite automata [God91] and data flow models [Bro91].

3.4 Tools

SDL models can be employed in several phases of the system development cycle, as noted in §3.1. Specific software tools support the developer in each of this phases. In this section we review the main types of tools for building and analyzing SDL models. The types of tools presented here can be found in several SDL tool frameworks, but they have common characteristics beyond framework or vendor specific issues.

- **Editors and semantic checkers**. SDL has a graphical syntax which makes the editor an important part of a tool framework. Editing is usually doubled by a syntactic and semantic checker, which ensures that the static semantics constraints of SDL are met by a specification.

- **Simulation tools** perform a symbolic execution of an SDL specification, conforming more or less to the formal dynamic semantics (§3.3.3). Simulation tools provide usual debugging functionality (step by step execution, breakpoints, investigation of system values, etc.) as well as more advanced features (stepping backwards, automatic stimulation of the system with signals, random simulation, tracking of complex conditions e.g. specified through a state machine, etc.).

  Because the real execution and communication times under simulation differ from those found in implementations, simulation tools typically use an artificial notion of time, and *control* time passage during simulation. Thus, the notion of time in simulation is more restrictive than the one specified by the formal semantics (§3.3.3). This issue is further discussed in §3.5.

- **Verification tools** can be used to prove formally that an SDL system specification satisfies a certain behavior property. The way the *property* can be defined is tool-specific, and can be a temporal logic or another formalism (Message Sequence Charts used as property specification language, automata based languages, etc.). The verification *methods* implemented by SDL tools are derived from *model checking* [QS82, CES86] (see also the monographs [CGP99] and [Hol91]).

  Dealing with time can be an important aspect of verification, and tools typically use a controlled notion of time equivalent to that used by *simulators*.

- **Code generators and deployment tools** allow the developer to obtain an implementation for a specific platform automatically from the SDL specification. As SDL is an imperative, design-oriented language, the translation of most SDL constructs into implementation objects is straightforward.

- **Test generators** can be used for automatically deriving test cases from an SDL model. In this case, the SDL specification is considered a correct, high-level description of the desired system functionality. Test cases corresponding to particular system executions are derived using simulation techniques. Current test generation tools and simulation tools handle system time in a similar manner. However, the test generators of which we are aware use the information concerning time only to correctly explore the SDL system execution, and do not generate timing information in the tests.

3.5 Discussion

This section complements the preceding SDL language description with some general remarks on the usability of SDL for the specification and analysis of real-time systems. The ideas outlined here are also elaborated in [BGK+00].

Specification vs. programming in SDL

SDL has the double aim of being on one hand a high-level *specification* formalism, which means it must abstract from certain implementation details, and on the other hand a *programming* (or *description*) formalism from which direct code generation is possible. The two roles of the language are sometimes conflicting, and in many cases the *description* side has been given priority.

In consequence, SDL has several characteristics which make it interesting as a *design language* for real-time systems: native asynchronous communication, timer constructs, hierarchical organization of the specification, etc. However, for requirements and high-level system specification, the constructs provided by SDL are mostly insufficient.

In [BGK+00, BGM+01] we proposed several extensions to SDL, necessary in order to capture descriptive information appearing in the initial phases of system modeling:

- Assumptions or knowledge about *channel reliability*, with attributes like loss rate, minimal/maximal delays, etc. As all other information types enumerated below, information about channels may be available early in the development cycle, and should be captured in the SDL model. It is useful during simulation and verification.

  Currently, in order to model such information in SDL, one has to model the behavior (losses and delays) of a channel in an imperative manner, e.g. through an SDL process. The approach has several drawbacks enumerated in [BGK+00], but is nevertheless used in practice whenever the characteristics of channels are essential for simulation and verification purposes (e.g. in specifications of flow control protocols, which are designed precisely to cope with losses and delays).

- Information about *execution times*, especially in abstract SDL models containing informal action specifications.

  Currently, execution times must be modeled by introducing explicit waiting (e.g. with timers). This works for specifying minimal or exact execution times, but cannot express maximal execution times. Using such programming concepts to model high-level descriptive information about timing also changes the meaning of the model, which can prevent it from being used for tasks such as code generation.

- Information about the *behavior of the environment*. In SDL, the system may communicate with the environment, which is completely unspecified. However, in real systems some characteristics of the environment, concerning the ordering and periodicity of signals, are frequently known. The well-functioning of the system may rely on such assumptions on the environment.

Some of these extensions and their possible exploitation are described in the later chapters of this thesis.

Reasoning about time in SDL specifications

The manner in which time progresses during the execution of a system is left unspecified in SDL. Z.100 [IT99b] specifies that an indeterminate amount of time may elapse during the execution of any action, and two different executions of the same action may take different amounts of time. Moreover, any agent may be kept waiting or may be suspended indeterminately by the system scheduler, which is not specified. The modeling of time in the formal ASM semantics [IT99c] complies with the above informal description.

These assumptions about the behavior of an SDL system are the *minimal* hypotheses that can be assumed about any implementation of the system. With such loose assumptions about the performance of the underlying machine, many unrealistic execution scenarios of an SDL specification are actually allowed by the semantics. The result is that it is difficult to guarantee almost any time-related property about the system behavior. This problem, previously raised by other authors [Boz99, MGHS96], is also examined in [BGK+00, BGM+01].

SDL simulation and verification tools solve this problem by deviating from the standard semantics in what concerns time. As mentioned in §3.4, in simulation and verification tools time is a "logical" parameter, controlled by the tool. The control is based on a set of tool-specific rules (which may be parameterized). For example, the *Object*GEODE simulator [TEL00a] controls time passage by considering that actions take 0 time to execute unless otherwise specified, and that time only passes when the system is idle (all agents are in a stable state waiting for an external stimulus). The solution provided by tools falls in the other extreme: it idealizes the performance of the underlying machine and may consider unrealizable certain realistic execution scenarios.

In Chapter 6 we propose a solution for this problem, based on constructs and techniques initially developed in the framework of timed automata. The idea is to include descriptions of assumptions on time in the SDL model, and to use these assumptions for a controlling time progress during simulation and verification.

Chapter 4

MSC and GOAL

A central aspect of the real-time systems specification and validation process adopted in this work is the specification of *timing requirements*. Such requirements may serve different purposes:

- during system analysis/specification, requirements describe features of the system on a high-level of abstraction,
- during validation, requirements describe properties of the system which have to be verified,
- for automatic test generation, requirements describe the typical functions of the system, for which test cases have to be generated.

Requirements may be described, up to a certain limit, in SDL. However, they are usually situated on a more abstract level for which SDL is ill adapted. For instance, a pure functional requirement for a typical behavior of a system, such as "the system responds to a signal A with a signal B within ... time units" is not concerned with the structure of the system, still the structure has to be described if the requirement is written in SDL. Moreover, as noted in the end of Chapter 3, SDL is in general ill adapted for non-imperative (i.e. declarative) description of behavior.

In practice, specific (declarative) languages are used to specify requirements. Such languages range from logic formalisms (first order logic, temporal logics) to automata-based languages or trace languages. Industrial practitioners show a preference for trace-based requirements languages, this being proved by the integration of such languages within modern analysis and design methodologies [IT97, OMG99].

The SDL methodology guidelines [IT97] recommends the use of MSC [IT99a] as requirements specification language in the context of designing SDL systems. We will examine also a second language, GOAL [ALH95], as it provides a complement to MSC. GOAL is defined and supported in the *Object*GEODE toolset [TEL00a], and is more suitable than MSC for the specification of properties employed in formal verification. In Chapter 7, MSC and GOAL are used and extended for expressing time-related properties of real-time systems.

4.1 MSC

Message Sequence Charts is a formal language for representing execution traces of systems in terms of the messages exchanged between the system *components* or with the *environment*. The MSC Language is standardized and maintained by the ITU–T as the Recommendation Z.120 [IT99a].

MSC emerged from a practical need to express system execution traces in a visually intuitive and precise way. Many non-standard precursors of MSC (time sequence diagrams, arrow diagrams, information flow diagrams, interworkings) were used locally in companies and standardization bodies in the telecommunication sector. The work on a standard language for representing execution traces begun in the ITU–T around 1990.

The first version of MSC dates from 1992, and it provides a textual and a graphical syntax, as well as an informal semantics. The elements defined in MSC-92 – instances, messages, conditions, actions, timers, process creation and process stop, coregions and sub-MSCs – remained essentially the same in the current version of the language (MSC-2000).

Around 1993, substantial work was put into defining a formal semantics for MSC. There were three main proposals: one based on automata [LL93, PL93], one based on Petri nets [GPR93], and one based on process algebra [MR94, MR96]. An improved version of the third one is currently part of the MSC standard (Z.120 Annex B).

The 1996 version of MSC added several structural concepts – inline expressions, MSC references and High-level MSCs (HMSC) – which facilitate the construction of large specifications. A few basic concepts were also added: gates and general ordering arrows.

The recent interest for expressing and analyzing *timing* constraints with MSCs lead to a series of new concepts, added in MSC-2000. This version also includes constructs for declaring and manipulating data.

Paradigm and scope

A *basic* MSC specification essentially describes a set of *instances* and *messages* exchanged between these instances. An MSC instance represents a component of the designed system, but the level of granularity is undefined: the instance may correspond to an agent, a set of agents or the entire system in an SDL specification.

Besides messages (outputs and inputs), other occurrences may be represented on an MSC instance: timers, actions, local and global conditions. The language offers mechanisms for building complex MSCs by composing basic MSCs.

The scope of MSC is the specification of requirements for reactive systems and system components. Such requirements may appear in the initial phases of system development (analysis, specification, design), they may be execution traces built for debugging purposes, or they may be a basis for simulation, verification and test case generation.

Language definition artifacts

The MSC language definition (Z.120, [IT99a]) includes a textual syntax (MSC/PR), a graphical syntax (MSC/GR), an informal semantics written in English and a formal semantics (Z.120 Annex B). The formal semantics is not stable yet in the current version of the language; consequently, in the following we refer to the formal semantics of MSC-96.

The textual and graphical syntaxes have equivalent power of expression. The graphical form is easily readable and it is the form in which documents are sketched and used by humans. The textual format of MSC was designed primarily for facilitating the electronic exchange of documents between CASE tools.[1] Being simpler and more consistent than the graphical syntax, the textual syntax is also used as basis for the definition of the formal semantics.

[1] Computer Aided Software Engineering tools

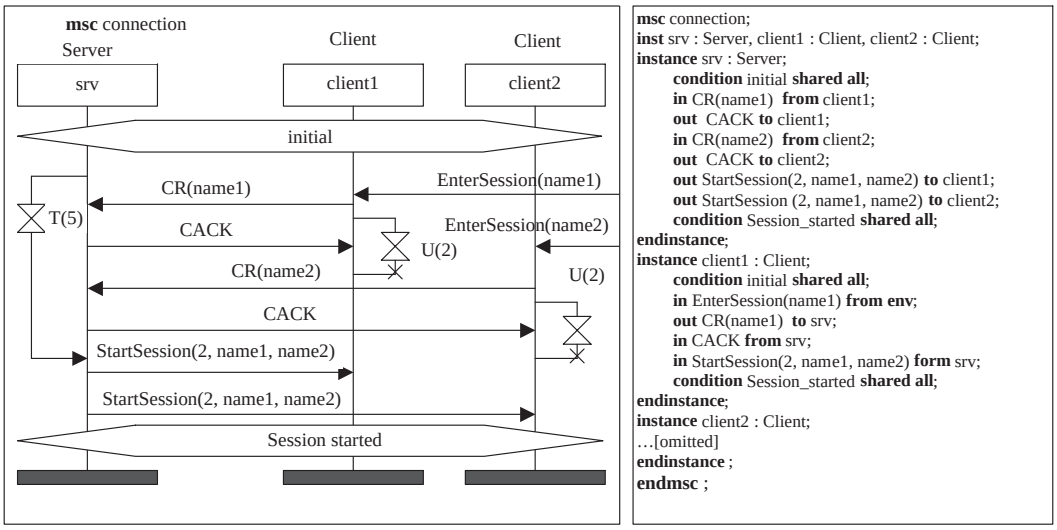


Figure 4.1: Basic MSC for the connection phase of the protocol

In what follows we will use a simple made-up example for presenting the MSC language constructs. The example is a simple protocol, belonging to the application layer in the OSI stack, in which several clients use a central server for exchanging services (the nature of which is not defined) in a session-based fashion. The initial connection phase (with 2 clients) is shown in Figure 4.1, in both graphic and textual format.

4.1.1 Basic MSC

A Basic MSC is a specification describing *instances* and *events*. The nature of these concepts is explained in the following paragraphs.

Instances, events, ordering

Instances are distinct sub-parts of a system, characterized by a *name* and a *type*. The type in MSC is just a name and it is supposed to be a meaningful information in the context of the language in which the system is modeled. For example the type may correspond in SDL to an agent type, agent instance set, etc.

As shown in Fig. 4.1, in graphical format, an instance is represented as a vertical bar, beginning at the top with a rectangle containing the name of the instance and ending at the bottom with an end symbol (or with a stop symbol as we will see later).

Various types of events may be represented on an instance. Fig. 4.1 shows message output and message input events (emission and reception of a message are considered distinct events, i.e. the model is asynchronous), timer events (set, reset, timeout) and global conditions.

The events drawn on an instance bar are considered *ordered* in time, from top to bottom. However, the global ordering of events from multiple instances is not necessarily the visually intuitive order. Actually, the *global order* of events is a *partial order*, defined by the local instance orders and the *causality* (a message must be emitted before it is consumed).

Thus, the MSC in Fig. 4.1 states that CR(name1) is received by srv before CR(name2), but formally it says nothing about the order in which these two events were emitted.

The *environment* of an MSC is capable of emitting and accepting messages. In the graphical form, the outer border represents the environment of an MSC, from and to which message lines may be drawn. In the textual form, the environment is denoted by env. For example, in Fig.1 the message EnterSession is received from the user, who is part of the environment.

Messages

A message represents an asynchronous communication occurrence between two MSC instances. The message has a *name* and may carry *data* parameters. In MSC, a message defines two events: the output and the input. The latter may actually represent either the *receipt* or the *handling* of the message by the destination instance, as the relation between MSC models and other models of a system (e.g. SDL) is outside the scope of [IT99a].

In the graphical form, a message is drawn as an arrow, which must be *horizontal* or heading *downwards*. The way the message is drawn does not imply anything about the delay between the emission and the reception of the message. However, the above rule is useful as it eliminates geometrically the possibility of cyclic causality.

Lost and found messages may also be represented on MSCs. A lost message defines an output event with no corresponding input. A found message defines an input event with no corresponding output.

In MSC-2000, the message types and the types of parameters may be declared. However, MSC does not describe a data definition language, so the user may use data types defined in other languages (SDL, UML, ASN.1, etc.). Such external data types are referenced through names that are not interpreted in the MSC semantics. Instances may own *variables* which allows for the specification of more complex requirements, such as dependencies between parameter values of different messages. Constructs for assigning a value to a variable may appear either in message receipts or inside actions.

General ordering arrows and coregions

General ordering arrows are useful when we want to constrain the order of two events, which are otherwise unrelated by the MSC partial order. For example, in Fig. 4.1, the emission of CR(name1) by client1 is not related in any way with the emission of CR(name2) by client2. To specify that client2 emits first, one would have to use a general ordering arrow.

Coregions are used to relax the local ordering of events on an instance. A coregion belongs to an instance and is defined by a starting point and an ending point. The effect of a coregion is that events represented inside the coregion boundaries may occur in any order and not necessarily in the visual order.

Timer operations

MSC timers are inspired from the homonym concept of SDL. An instance may specify a timer *set* followed either by a timer *reset* or by a *timeout*. These timer events may be represented also individually, when the description of an instance is split into more MSCs, and the timer constructs appear on different MSCs. A timer set by an instance may only timeout on or be reset by the same instance.

As noted in [BAL97], timers may be used for expressing either minimal or maximal delays between events on a same instance. In our example in Fig. 4.1, the timer T on the instance srv models the requirement that the server must wait for at least 5 time units for incoming connections, and after that it may consider that subscription phase ended. Timer U on client1 and client2 models the requirement that the connection acknowledgement must come from the server in at most 2 time units since the connection request was sent.

Timer operations provide a very restrictive mechanism for specifying timing constraints, for the following reasons: firstly, timers are just discrete events in the formal semantics of MSC, with no special timing connotations. Secondly, there are constraints which cannot be represented with timers, such as the delay between the emission and the reception of a message.

For this reason, additional timing annotations were introduced in MSC-2000. They are discussed in a further section.

Conditions

An MSC condition construct (e.g. initial or Session started in Fig. 4.1) represents either a significant state of an instance, or a state shared by several instances. Shared states (conditions) are useful as they provide synchronization points between instances: a shared state introduces a *single event* which is shared by all instances and thus constrains the global partial ordering of events.

A condition is characterized only by a *name*, and it does not necessarily say something about the actual state of the system in terms of variables, message queues, etc.

Actions, Method Calls, Instance Creation, Stop

Actions – containing variable assignments or uninterpreted text – may be represented on instances.

A concept of method call similar to the SDL RPC was introduced in MSC-2000. A method call begins with a message between two instances, representing the initiation of the method, and ends with another message representing the reply. Between the two, a *suspension* region is drawn on the sender, and no message, timer or other construct may appear in this region.

Creation of an instance by another instance, as well as termination of the execution of an instance may be represented as an MSC event.

4.1.2 Structuring concepts

The MSC language defines several mechanisms for structuring complex specifications and describing non-linear control flows. These are: instance decomposition, inline MSCs, MSC references and High-level MSCs (HMSC).

Instance decomposition

An instance from one MSC may be refined in another MSC containing sub-entities of that instance. If an instance I from an MSC M1 is decomposed in another MSC M2, then the environment of M2 will send and receive exactly the same messages that are received and sent by I in M1. A similar rule applies for timer events: a timer event appearing on the decomposed instance I in M1 must appear someplace on an instance in M2.

Inline MSCs

Inline expressions allow to express non-linear control flows inside an MSC. They are based on the notion of MSC composition operator. The operators are:

- alternative choice between two or more MSCs (alt operator),
- parallel execution of two or more MSC sections (par operator),
- repeated execution of an MSC section (loop operator),
- optional execution of an MSC section (opt operator)
- execution of an MSC section with an option for treating exceptions (exc operator).

On the semantics of inline operators, we note that the boundaries on an operand MSC are not considered synchronization points for the instances involved in the MSC. Thus, for example, if two instances I1 and I2 are described by a loop MSC M, this is equivalent to the repetition of M an arbitrary number of times, but does not introduce synchronization points between two successive occurrences of M. For a more complete definition, the reader is referred to [IT99a].

Gates, MSC references and HMSC

MSC *references* may be used to refine the behavior of one or more instances from an MSC in another MSC. A referred MSC must contain the same instances as the referring MSC. An advantage of using MSC references is that they can be parameterized with data.

Gates are used to clarify the connections of a referenced MSC when it is put in a larger context. They are used to provide a mapping between the environment (bounding box) of the referenced MSC and the instances or the environment of the referencing MSC.

Gates are inspired from the homonym concept from SDL; thus, a gate is a named interface between an MSC and its environment. Every message or general ordering arrow coming from the environment or going to the environment comes or goes through a gate. The gate through which a message is transferred may be declared explicitly, or introduced implicitly (with a default name depending on the name and direction of the transferred message).

Gates and MSC references allow for the description of *High-level MSCs* (HMSC). A HMSC is a graph formed of start nodes, end nodes, conditions, MSC references and connection points. Arrows in this graph represent the flow of control. Multiple arrows outgoing from the same node represent alternatives. Conditions represent synchronization points for all the instances concerned by the HMSC. Strict static requirements for well formedness are given in the language definition, such as: there should be exactly one start node and one end node for each HMSC. HMSCs allow a graphical representation of structured MSCs but the power of expression is the same as that of textual composition operators (alt, loop, par, opt). An example of HMSC is given in Fig. 4.2; it includes references to MSC that are not described here for brevity.

As for inline composition operators, we note that the sequential composition in HMSC does not introduce implicit synchronization points between the instances concerned by the HMSC. The semantics of the sequential composition of two MSCs is equivalent to that of the juxtaposition of events from the two MSCs on respective instances.

4.1.3 Semantics and decidability

As in the case of SDL, there were multiple attempts for defining a formal semantics for MSC, based on various formalisms. The problem is easier than in the case of SDL, because MSC

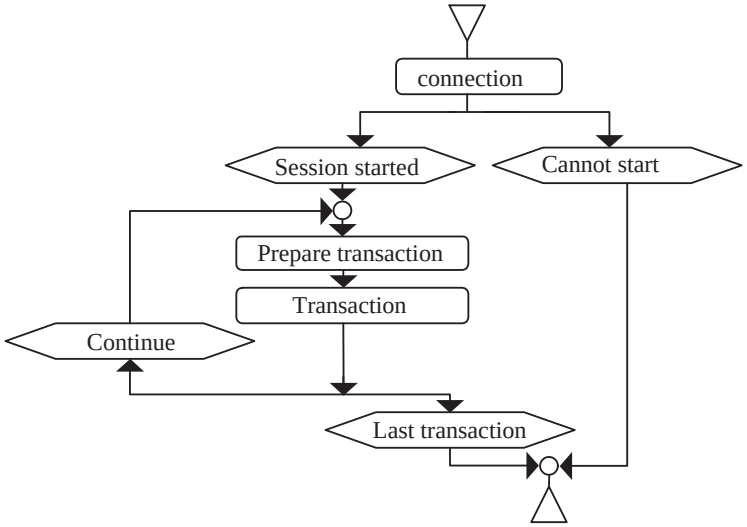


Figure 4.2: Example of high-level MSC

is a language for describing *traces* and a formal semantics must only provide a mathematical relationship between a well formed MSC and the corresponding set of acceptable traces.

The language of event traces described by a Basic MSC is a regular language, and there are several ways in which it can be characterized formally: using finite automata [LL93, PL93], 1-safe[2] Petri-nets [GPR93], or process algebra terms [MR94]. The three approaches mentioned above are briefly presented in the following paragraphs.

The language of traces defined by a High-level MSC is no longer a regular language. This raises decidability issues which are discussed in the end of this section.

Petri-net semantics of Basic MSC

A Petri-net based semantics for MSC is described by Grabowski et. al. in [GPR93]. Fig. 4.3 shows an example of a simple Basic MSC and the corresponding labeled Petri net. We employ the usual notation for Petri nets, in which circles denote places, rectangles denote transitions, and arcs denote token flow.

A *place* represents either the state of an instance between two consecutive MSC events, or a message that was sent and waits to be received. The labels on places correspond to their function (in our example, labels are shown on places representing waiting instances). *Transitions* represent MSC events. The initial marking puts one token in each place corresponding to the beginning of an instance (except for dynamically created instances).

Shared MSC *conditions*, which provide synchronization between instances, are modeled using a unique synchronizing transition in the Petri net. A condition shared by n instances is represented as a transition with n input arcs and n output arcs, each corresponding to one of the n instances.

It can be easily shown that Petri nets constructed from Basic MSCs following the above rules satisfy the 1-safeness requirement (i.e. have at most one token in each place at any time). It

[2]The meaning of 1-safeness is that in all reachable markings of a net, the number of tokens in any place never exceeds one. See also [Pet81].

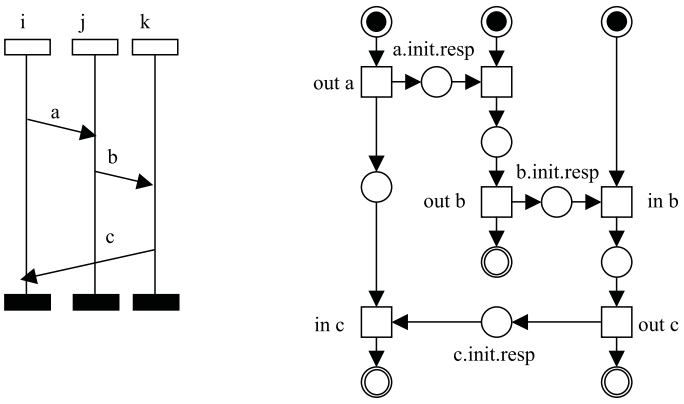


Figure 4.3: Labeled Petri net corresponding to a Basic MSC

follows that the language of traces defined by the MSC is regular, since the marking automaton of the Petri net is finite.

Automata semantics of Basic MSC

Another semantics for MSC, based on automata, was proposed by Ladkin and Leue [LL93, PL93]. The automaton they construct from an MSC specification is roughly equivalent to the marking automaton of the Petri net proposed in [GPR93].

Process algebra semantics of Basic MSC

The standard MSC formal semantics (Z.120 Annex B) is an elaborated version of the process algebra-based semantics of Mauw et. al [MR94, MR96]. The semantics of an MSC is given by a term in a process algebra PA_{BMSC}, which is an extension of the PA_ϵ defined in [BW90]. The signature of PA_{BMSC} contains:

- Empty process (ϵ) and deadlock (δ) symbols.
- Action constants, which are labels denoting MSC events: actions, outputs, inputs, etc.
- Operators for alternative composition ($+$), sequential composition ($\cdot$), free merge ($\|$), left merge ($\lfloor\lfloor$) and termination ($\surd$).

For a more complete description of the process algebra semantics of MSC, the reader is referred to [MR96] and to Z.120 Annex B.

Decidability of HMSC

As defined by the standard, the *upper* and *lower* boundaries of Basic MSCs referenced from a HMSC *do not* constitute *synchronization* points between the represented MSC instances. This means that, if we have a sequence of two Basic MSCs containing two instances (A and B), there may be traces represented by the MSC in which events appearing in (the beginning of) the second MSC on instance A, occur before events appearing in (the end of) the first MSC on instance B.

If we consider the Petri net semantics of MSC, which is quite intuitive for this example, the effect of this lack of synchronization at MSC boundaries destroys the 1-safeness property of the resulted Petri net.

Indeed, if we take a Basic MSC M containing only two instances A and B and a message m from A to B, and a HMSC which describes an infinite loop on the BMSC M, the Petri net corresponding to the HMSC contains places for which the marking may increase to infinity. This corresponds to the situation where A sends the message m continually at a higher rate than B can consume.

Because of this choice of the semantics of HMSCs, most model checking problems for HMSCs are undecidable. An important example is the undecidability of the problem of *emptiness of the intersection* of two HMSCs. For a survey of the undecidability problems of HMSCs, the reader is referred to [MP00].

In order to overpass the undecidability problems mentioned above, validation tools using MSC as property specification language employ a different semantics for HMSCs. For example, the *Object*GEODE tool on which our further work relies, considers that Basic MSC boundaries introduce synchronization points, and thus a HMSC defines a regular language of traces and is equivalent to an automaton. Moreover, the tool does not allow free-formed HMSCs, but uses operators similar to the inline MSC operators described on page 66.

4.1.4 Tools

MSC models can be employed in different phases of the system development cycle, as noted in the beginning of §4.1. Various software tools provide support for each of these phases. We enumerate below some of the tool types which are involved in building and manipulating MSC descriptions, in order to set the background for the work presented in subsequent chapters of this thesis.

- **Editors, syntactic and semantic checkers**, used for manual editing of MSC specifications and static checking.
- **Simulation and verification tools** for other system models, such as SDL models. They use MSC as an auxiliary language, and may take MSC specifications as input, or produce them as output.

  In input, an MSC specification can be used for guiding a simulation, or for verifying that a system model is compliant to a requirement specified by the MSC. The nature of the compliance relationship is a tool specific issue; for example, an SDL specification may be considered compliant to an MSC either if there is an execution of the SDL specification which produces a trace that can be found in the MSC (possibly modulo some unobservable events) or if all executions of the SDL system produce traces from the MSC. This kind of functionality is frequently provided by SDL simulation and verification tools.

  In output, MSC may be produced automatically to represent debugging information. This is also a frequent functionality of SDL tools.
- **Test generation tools** which generate tests either directly from an MSC, or from an MSC and another formalism, such as SDL.

  The MSC language is suitable, within a certain extent, for the specification of test cases for asynchronous reactive systems. The latest version of the ISO test specification language TTCN [ETS00] actually defines MSC as an alternative representation for test cases.

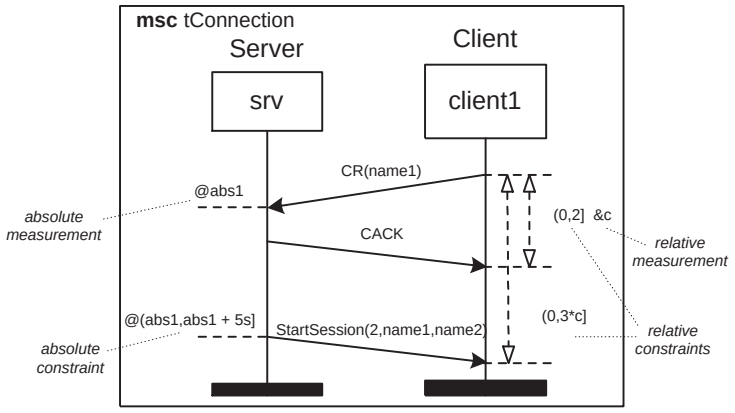


Figure 4.4: MSC with timing annotations

In test generation tools, MSC may be used either in output, to represent the test cases, or in input, to represent abstractly a test purpose for which test cases have to be elaborated based on another system model (e.g. SDL).

4.1.5 Specifying timing information

As a result of an identified need in real-time systems development, the specification of *timed* event traces was approached in MSC-2000:

- The *semantics* of MSC-2000 was adapted so that an MSC specification describes a set of *timed traces* of the following form: $(e_1, t_1, e_2, t_2, e_3, t_3, ...)$. In this trace $e_1, e_2, e_3, ...$ are discrete events (outputs, inputs, actions, timers, etc.) and $t_1, t_2, t_3, ...$ are relative time durations between successive events. At the time of writing, the timed formal semantics of MSC-2000 is not yet stable.
- Descriptive timing (*constraints*) may be introduced in MSC specifications. Their purpose is to specify the possible values for the time projection of a trace (i.e. $t_1, t_2, t_3, ...$). The annotations may specify either the absolute time of occurrence for an event, or the relative delay between two arbitrary events.

  Both time of occurrence and delays can be specified using a (possibly degenerated) interval in the domain of `Time` values (which is not specified but can be assumed to be that of positive reals, $\mathbb{R}_+$)
- *Measurements* allow to obtain (and store in a variable of type `Time`) the relative delay between two events, or the absolute time of occurrence of an event. The measured time may subsequently be used in a *constraint*.

The MSC in Fig. 4.4 shows the representation of timing measurements and constraints. These mechanisms can be used both in Basic MSCs and in High-level MSCs; in the latter they are attached to the beginning or the end of a referenced Basic MSC, and refer either to the first or to the last event in that BMSC.

Note also that in the timed semantics of MSC defined informally in the standard [IT99a], no particular timing interpretation is attached to *timer* constructs. Thus, timer

sets/resets/timeouts model discrete events and do not imply anything as to the timing of their occurrence.

The introduction of timing information in MSC raises the problem of the *internal timing consistency* of an MSC specification: "can there be any trace satisfying the timing requirements given in the MSC, or are inter-event delays contradictory?". This problem has been studied before, and static consistency analysis solutions based on graph theory are given in [AHP96] and [BAL97].

The model checking problem for timing properties specified with MSCs has not been extensively explored previously. In Chapter 7 we discuss the manner in which MSCs with timing annotations may be used to specify and verify quantitative temporal properties of real-time systems modeled in SDL.

4.2 GOAL

Scope

GOAL [ALH95] is the requirements specification language supported by the *Object*GEODE toolset [TEL00a]. It originates in the observer language of the Veda tool, described in [Gro89]. Its scope of applicability is more reduced than that of MSC, in the sense that its graphical representation is not suited for capturing high-level requirements during the analysis and design phases. GOAL is used for:

- *Automatic verification* of properties on SDL models. GOAL provides additional functionality compared to MSC in this area, as it can describe properties referring to the internal structure (agents, states) of an SDL system, or to the value of internal data. It also defines a clearer meaning for property satisfaction.
- *Guiding simulations and the verification process* by: modeling the behavior of the environment, cutting the exploration of parts of the model, injecting faults, unexpected signals, etc., and producing customized traces and statistics.

In this work we are interested in GOAL primarily as a property specification language. In later chapters of the thesis, we discuss extensions of GOAL for expressing timing properties of real-time systems, as well as timed property verification methods and tools.

Paradigm

GOAL is an automata-based language. A GOAL specification, called *observer*, is an extended finite automaton designed to be executed synchronously with an SDL specification, during simulation or verification. The meaning of synchronicity is that from synchronized composition of automata.

The *transitions* of the observer are triggered by events occurring in the SDL model: transmission or reception of signals, firing of transitions, creation or stopping of processes, time progression, etc. The *states* of the observer are classified into *ordinary*, *error* or *success* states which correspond to property satisfaction or breaking.

The following sections describe the language concepts and the execution model of GOAL.

4.2.1 Language concepts

Structure

A GOAL observer specification is similar to the behavior description of an SDL agent. The differences are:

- An observer is a stand-alone state machine, it is not connected through signal routes or channels with other state machines.
- Besides internal data (variables), an observer may declare and use *probes*. Probes are access paths to SDL model entities, which can be used to observe or modify SDL model objects.
- The transitions of a GOAL observer are triggered either by a **provided** clause (similar to SDL continuous signals, see §3.2) or by a **when** clause which observes events happening in the SDL model (see next paragraph). **Input** clauses are not allowed, as GOAL observers do not communicate through signals. Additionally, in GOAL, **provided** clauses *do not* have lower priority than **when** clauses.
- The transitions *must not* contain **output**, **set**, **reset**, **create** or **stop** actions, as well as several expressions defined in SDL (**self**, **parent**, **offspring**, **sender**).
- The observer state machine must be *deterministic*, i.e. two transitions should not be simultaneously enabled in a state, and transitions must not contain informal decisions or *anyvalue* expressions. This is a practical rule imposed by the simulation and verification tools to simplify state space exploration.
- **Task** actions on transitions can be used to modify both observer variables and SDL model objects (SDL variables, signal queues).
- In order to describe a property with an observer, some of its control states can be declared as *error* or *success* states.

Observation mechanisms

When clauses may be used to observe the following types of discrete events occurring in the SDL model:

- the firing of a particular transition,
- transmission or reception of signals,
- creation or stopping of processes,
- procedure calls.

Additionally, **provided** clauses combined with *probes* may be used to observe the values of SDL model variables, as well as time progression (by testing the value of **now**).

GOAL observers may be combined with a transition filter mechanism, so that the observer may cut the exploration of a part of the state space. This is an effective state space reduction mechanism, when a part of the state graph is not interesting for the verified property.

Fig. 4.5 shows an example of GOAL observer for the session oriented protocol introduced in §4.1.1. It specifies that a successful initialization must take at most 5 time units, from the moment the initialization phase begins, until the sending of the first StartSession signal. The representation of **when** clauses is shown in Fig. 4.5.

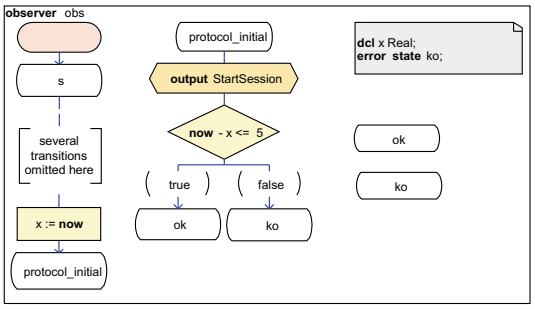


Figure 4.5: An example of GOAL observer

4.2.2 Observer execution

Unlike MSCs which are descriptive, GOAL observers are executable specifications. In order to check a property of an SDL system, an observer is executed by a simulation or verification tool in parallel with the system.

The *Object*GEODE simulation/verification tool, which implements GOAL, uses a notion of simulation step which normally corresponds to the execution of an SDL transition. A series of observable events (of the kinds described in the previous section) may occur in such an execution step. These events are tracked by the tool, and after executing the SDL step the following operations are repeated for each tracked event:

1. The observer's fireable transition clauses are evaluated, according to:
   - the event being processed,
   - the global state of the model, after execution of the simulation step.
2. If the observer does not have any fireable transitions, its state remains unchanged and the next event is processed.
3. If the observer has more than one fireable transition, a dynamic error occurs (since the observer must be deterministic) and observer processing stops.
4. If the observer has only one fireable transition, it is executed. If execution fails (due to a dynamic error) then observer execution stops. If execution is completed successfully, the next event is processed.

A simulation step (e.g. firing a SDL transition) generating several observable events may lead to several transitions of a same observer.

Observers can be used to check automatically that a model behaves correctly, that is to say that it meets user-defined behavior constraints. To describe a property with an observer, its control states must be classified as ordinary, *error* or *success* states. The observer should move to a success state whenever the expected property is met, and to an error state whenever an unexpected behavior is observed.

Usually, *error* states are observer state machine sink states. Since the observers are always executed in parallel with the model, when an observer reaches an *error* (respectively a *success* state), the property that it verifies is false (respectively true) for the scenario that the simulator has executed from the initial state up to the current state.

This mechanism is sufficient for verifying *safety* properties using GOAL observers. *Liveness* properties may also be checked using GOAL observers, in a special verification mode (*liveness mode*) of the *Object*GEODE tool.

In *liveness* mode, observers are executed synchronously with the SDL model in the same way, but they are regarded as Büchi automata [B60]. The verifier looks for infinite executions (loops) which do not contain *success* states (i.e. *success* states are viewed as *progress* states, and the tool searches non-progress cycles).

4.2.3 Specifying timing properties

There are no specific constructs for measuring time passage in GOAL. As an observer is always executed in parallel with an SDL specification, it may use the value of **now** to observe time progress. This could normally suffice to express infinitely complex timing properties.

In practice, due to the manner the *Object*GEODE tool manages time, **now** is of no use in GOAL specifications in verification mode. In Chapters 7 and 8 we describe several extensions of the GOAL language and of the verification tool, which make them suitable for the verification of quantitative temporal properties.

4.3 Expressivity of MSC and GOAL

MSC and GOAL have slightly different scopes: the former is more descriptive and oriented towards analysis and requirements capturing, the latter is more verification-oriented. However, both languages may be used in formal specification of properties and verification. In this section we take a brief comparative look to the power of expression of the two languages, viewed as property specification formalisms. We distinguish two comparison axes: the first measures the event observation facilities, the second measures the power of the underlying semantic model of the two languages.

4.3.1 Observation and other language facilities

GOAL is more powerful than MSC in what concerns the alphabet of events that may be observed. MSCs observe the following event kinds:

1. message outputs/inputs,
2. timers set/reset/timeout,
3. procedure calls,
4. process creation/termination
5. actions and conditions - existing SDL/MSC tools cannot actually map these to observable events in the SDL system, so these events are never observed.
6. time progress - existing SDL/MSC tools do not support this feature.

Additionally to these, GOAL is able to observe:

1. firing of SDL transitions,
2. time progress, effective in current SDL simulation tools but not in verification,
3. data values, discrete states of SDL agents, contents of signal queues.

MSC has the advantage of being more abstract and visually intuitive, while GOAL has the advantage of being a complete imperative language. One can write real event-driven programs in GOAL, with more complicated control flows than what can be expressed in MSC.

Additionally, GOAL programs may modify the simulated SDL model. This can be used for example for fault injection, or forcing the execution of certain parts of the SDL specification.

4.3.2 Semantic model and satisfaction relationship

In §4.1.3 we noted that the semantic model of standard MSCs is richer than finite automata, but most model checking problems are undecidable for this model. In practice, verification tools based on MSC, such as *Object*GEODE, use a semantics for MSC equivalent to finite automata. Thus, the basic semantic models of GOAL and MSC are identical.

The satisfaction relationship between a model and an MSC specification is outside the scope of the MSC standard, and depends on the choice of verification tools. Here are some examples of choices that can be made by tools:

- The MSC represents successful scenarios vs. error scenarios.
- All executions of the system must comply to the MSC vs. at least one execution must comply to the MSC.
- The MSC represents a *complete* trace (all observable events are represented) vs. the MSC represents a *partial* trace (additional events may occur between the events specified by the MSC).

In practice, no MSC tool gives control over all these parameters of the semantics of the satisfaction relation. Expressing some things (e.g. a combination of *complete* and *partial* scenarios in the same MSC) may not even be possible without some extensions to the language.

In contrast, the satisfaction relationship between SDL models and GOAL observers, using explicitly defined *success* and *error* states, is sufficiently flexible to encompass all the choices mentioned above.

4.3.3 Conclusion

To conclude, GOAL is more suitable than MSC for formally specifying properties of SDL models, when automatic verification is aimed. Nevertheless, MSC has several advantages: it is standardized, used on a larger scale (and for more various tasks), more abstract, simpler and more intuitive. For these reasons, in the context of this thesis we study ways to make both languages more appropriate for the verification of quantitative temporal properties of real-time systems.

Chapter 5

Timed automata

In the previous chapters we have presented the functional description languages on which the real-time specification and validation approach proposed in this work is based. There, we have outlined a series of lacks of these formalisms, which concern both the *specification* of timing information and the possibilities to *use* such information e.g. in formal reasoning about timing properties.

In this chapter we examine the *timed automata* model, introduced by Alur et al. [ACD93, AD94], which allows both the description of timing information and the formal (possibly automated) reasoning based on this information. In later chapters, we will use concepts and analysis techniques from the timed automata framework in order to enhance the support of the SDL-centered framework for the design and validation of real-time systems.

In the beginning of the chapter we discuss some of the choices that have to be made when explicit timing is introduced in a formalism. In §5.2 we introduce labeled transition systems (LTS) which provide the semantic basis for many formalisms including timed automata. We continue in §5.3 with the definition of the timed automata model, whose semantics is based on LTS. In §5.4 we examine the reachability problem for the timed automata model, and some analysis methods for solving it. The abstractions used for deciding reachability are useful for solving other important problems for timed automata, e.g. various model checking problems. We close the chapter with a discussion of the extensions of the timed automata model that have been studied in the literature.

5.1 Reasoning about time

A characteristic of real-time systems is that their correct functioning depends on the timing or their actions and responses. A real-time system model must include such timing information, representing either *requirements* or *knowledge* about the system behavior.

For validation purposes, including timing information in the system model is necessary but not sufficient. What is further needed are techniques for manipulating this information and deriving additional properties about the temporal behavior of the model.

Model-based vs. axiomatic frameworks

Various frameworks for reasoning about time have been proposed in the literature. There are two major lines of thought. One of them is concerned with deriving timing properties based on *behavioral models* (the model-based approach). The idea is to take the behavioral model of the

system, which is used for modeling and verifying functional properties, and to annotate it with timing information and use it for modeling and verifying timing properties. The techniques used for analyzing such models are based on the exploration of the model *state space*, and on methods derived from *model checking*. *Temporal logics* may be used in conjunction to these models to express properties.

Representatives of the model-based based approach are formalisms such as timed automata [ACD93, AD94], temporal extensions of Petri nets [Sif77, Ram74, MF76], temporal extensions of process algebras [NS91], etc. For property specification, they use real-time extensions of temporal logics (see for example the survey [AH91]), or automata-based formalisms such as timed Büchi automata.

The second category of frameworks for reasoning about time aim at modeling only the temporal properties of a system, independently of any behavioral model. Reasoning is possible based on a proof system, formed of axioms and inference rules, in which new properties may be derived from existing ones. Examples of such frameworks include *duration calculus* [CHR92] or timed extensions of Hoare logic [Sha95].

In this work we concentrate on a model-based approach, as we aim to support the validation of quantitative temporal properties based on (extended) SDL models. We take *timed automata* as starting point, because they are extended versions of finite automata, and thus semantically related to SDL.

Discrete vs. continuous time

Timed models can be classified in two categories: *discrete* models and *continuous* models. In *discrete* time models, time passes in discrete steps, so the distinguishable moments in the functioning of a system may be mapped on the set of positive integers. Any system event occurs at one of these countable moments.

In *continuous* time models, such as the *timed automata* model examined in this chapter, time is real-valued. Time passes continuously between two events occurring at moments t_1 and t_2, so other events are allowed to occur at any moment in the interval $[t_1, t_2]$. Continuous time models are also called *dense* time models.

From the point of view of the power of expression, the two classes of models are not equivalent: *continuous* time models are strictly more expressive than *discrete* time models. This is argued informally in [Alu91, AD94] and more formally in [HMP92, AMP98]. For example, [AMP98] shows that for a certain class of digital circuits modeled with timed automata, discrete time semantic misses a subset of the intended behavior.

From the point of view of the analysis techniques applicable to them, the two classes of models are quite different. On one hand, in discrete models time may be considered just another discrete variable of the system. Therefore analysis techniques for untimed models may be easily adapted to discrete timed models, with all the consequent advantages. On the other hand, continuous time models generate uncountable state spaces, so their analysis techniques *must* rely on a symbolic representation of time. With symbolic techniques, a possibly infinite set of explicit states is represented in one symbolic state using some coding method.

The use of symbolic techniques for handling time may create an overhead at analysis, so continuous models are generally regarded as more expensive than discrete models. However, the reverse is also possible, as enumerative techniques for discrete time models may suffer a state space explosion phenomenon in case of time-nondeterministic specifications. Several examples supporting this statement may be found in the case studies section of [Tri98].

Reasoning about time in SDL vs. timed automata

SDL contains constructs for describing time-driven behavior: the designer can use timers or enabling conditions involving the time variable **now** in order to describe such behavior. Thus, the execution of an action may be triggered or conditioned by time.

While these constructs can model infinitely complex behavior, SDL has two major drawbacks when one wants to *verify* temporal properties of timed systems:

1. The formal semantics of the language [IT99c] is loose about time progress: indefinite amounts of time may pass while a process is in a state even if it has a valid input signal waiting in the queue, and actions take indefinite times to execute. The only system component which behaves strictly with respect to time is the underlying component responsible for keeping track of timers and sending timer expiration signals. With such loose assumptions about the performance of the underlying execution machine, it is difficult to guarantee almost any time-related property about the system behavior. This problem was pointed out in [BGK+00, BGM+01].

2. The complexity of conditions on **now** is not limited. The modeler may describe indefinitely complex behavior, for which it is difficult to conceive analysis methods and algorithms.

Timed automata cope with both problems mentioned above:

1. They provide stronger requirements on time progress, which can be constrained by the state of the automaton. Thus, one can specify actions that occur at a specific moment or within a bounded time, unlike in SDL.

2. Time conditions can only have simple forms. As we will see in later sections, in timed automata the only mechanism to measure time is the clock. An automaton may use several clocks at a time, all of which progress at the same rate and can be initialized and tested separately. Time conditions are represented by conditions on clocks, which can only have some restricted form. These restrictions are essential to make it possible to solve analytically a series of problems on timed automata, such as the reachability problem or various model checking problems.

For timed automata techniques to be applied to SDL, a SDL specification has to conform, in a way, to the restrictions mentioned above. We discuss the implications of this in Chapter 6.

5.2 Labeled transition systems

Timed automata (TA) are a special kind of Extended Finite State Machines (EFSM), with specific means for describing *time-related behavior*. EFSM is a generic name for the class of models which are based on a finite state machines and are extended with additional capabilities such as variables. Various types of EFSMs are used for modeling the behavior of reactive systems, and they constitute the target model for analysis techniques such as model checking [QS82, CES86]. A common characteristic of all types of EFSM, including timed automata, is that their semantics is given as a (possibly infinite) labeled graph of *states* and *transitions* (Labeled Transition System – LTS).

Definition 5.1 (Labeled Transition System) *A* labeled transition system *(LTS) is a tuple* $(Q, Q_0, \Sigma, \rightarrow)$ *where* Q *is a set (states set),* $Q_0 \in Q$ *is a non-empty subset of* Q *(initial states),*

Σ is an alphabet of symbols (transition labels) and $\rightarrow$ is a ternary relationship on $Q \times \Sigma \times Q$. We denote $x \xrightarrow{a} y$ the fact that $(x, a, y) \in \rightarrow$.

The structure or content of labels is not defined at this level. Usually, when an LTS is used as the semantic model for a higher-level formalism, labels correspond to actions, transitions, etc. described in that formalism. For specific purposes (abstraction, verification) sometimes it may be useful to classify labels according to various criteria, e.g. observable/internal, or input/output/internal.

Semantics of LTS

From an operational point of view an LTS may be viewed as an automaton. Its execution begins in the initial state. The LTS is in one state at any time, and it may take a transition out of this state depending on the external constraints expressed in terms of labels. LTSs differ from finite automata in that they have no final state and acceptance conditions.

Definition 5.2 (traces and runs) *Let $S = (Q, Q_0, \Sigma, \rightarrow)$ be an LTS and $\varphi = (a_0, a_1, \ldots)$ a finite or infinite sequence of labels from Σ. φ is a* trace *of S iff there exists a sequence of states $\psi = (q_0, q_1, \ldots)$ so that $q_i \xrightarrow{a_i} q_{i+1} \forall i.\, i < |\varphi|$. The couple (φ, ψ) is called a* run *and is represented as $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$*

Depending on the purpose served by an LTS, we may consider different semantics for it. For example, if we want to check whether a certain trace is possible or not in an LTS S, we might consider the semantics of S as the set of all traces accepted by S. However, if we want to check that there are no sink states in an LTS, this semantics is not sufficient. We give below the definition of the strong equivalence relationship between two LTS (derived from the strong bisimulation relation of [Mil80]).

Definition 5.3 (strong equivalence) *Let $S_1 = (Q_1, Q_{0\,1}, \Sigma_1, \rightarrow_1)$ and $S_2 = (Q_2, Q_{0\,2}, \Sigma_2, \rightarrow_2)$ be two labeled transition systems. S_1 and S_2 are strongly equivalent iff there is a relation $\approx \in Q_1 \times Q_2$ such that:*

$$\forall q_1 \in Q_1. \forall q_2 \in Q_2.\ q_1 \approx q_2 \Rightarrow \begin{cases} \forall a \in \Sigma_1.\ \forall q_1' \in Q_1. q_1 \xrightarrow{a} q_1' \Rightarrow\ \exists q_2' \in Q_2 \text{ such that} \\ \quad q_2 \xrightarrow{a} q_2' \text{ and } q_2 \approx q_2' \text{ , and} \\ \forall a \in \Sigma_2.\ \forall q_2' \in Q_2. q_2 \xrightarrow{a} q_2' \Rightarrow\ \exists q_1' \in Q_1 \text{ such that} \\ \quad q_1 \xrightarrow{a} q_1' \text{ and } q_1 \approx q_1'. \end{cases}$$

and
$\forall q_{01} \in Q_{0\,1}.\ \exists q_{02} \in Q_{0\,2}$ *such that* $q_{01} \approx q_{02}$ *, and*
$\forall q_{02} \in Q_{0\,2}.\ \exists q_{01} \in Q_{0\,1}$ *such that* $q_{01} \approx q_{02}$ *.*

LTS as semantic model

LTS is a natural, although low-level model for representing computations in a Von Neumann architecture. Thus, a sequential program in a usual imperative programming language can be seen as an LTS. The states of the LTS are tuples of the form $\langle IC, V \rangle$ where IC is the instruction counter that keeps the position of the next instruction to be executed by the program and V is

the vector containing the values of all variables in the program. If some variable domains in the model are infinite (as it is the case with clocks in TA) the corresponding LTS may not be finite.

Concurrent programs such as SDL specifications may be represented with LTSs as the set of all possible interleavings of concurrent actions. For that, some level of atomicity has to be assumed for the instructions of the concurrent programs. The standard ASM semantics of SDL [IT99c] for example is in line with this requirement, and is equivalent to an LTS-based semantics.

Generally, when LTS are used as a basis for the semantics of a high-level formalism, the definition is done on several layers so that the actual LTS corresponding to a specification in the initial formalism is not defined explicitly. In the case of SDL, the ASMs semantics provides an intermediate level between an SDL specification and the corresponding LTS. Along the same line, the LTS corresponding to a concurrent model is usually not defined explicitly; instead, the LTS is built from other LTSs corresponding to sequential components of the model, by using a suitable LTS composition operator.

LTS composition operators

In this section we show several composition (product) operators that are usually employed to model concurrent programs. In the following definitions, let $S_1 = (Q_1, Q_{0\,1}, \Sigma_1, \rightarrow_1)$ and $S_2 = (Q_2, Q_{0\,2}, \Sigma_2, \rightarrow_2)$ be two labeled transition systems.

The *asynchronous product* models non-synchronized parallel composition of programs, i.e. arbitrary interleaving of actions from the two programs.

Definition 5.4 (asynchronous product) *The* asynchronous product *of S_1 and S_2 is the LTS $S_1 \| S_2 = (Q_1 \times Q_2, Q_{0\,1} \times Q_{0\,2}, \Sigma_1 \cup \Sigma_2, \rightarrow)$ where $\rightarrow$ is defined as:*

$$(q_1, q_2) \xrightarrow{a} (q_1', q_2') \quad \textit{iff} \quad \begin{cases} q_1 \xrightarrow{a}_1 q_1' & \textit{and} \quad q_2 = q_2' \ \textit{or} \\ q_2 \xrightarrow{a}_2 q_2' & \textit{and} \quad q_1 = q_1' \end{cases}$$

Alternatively, the asynchronous product may also be defined such that the two LTS can both take a step at the same time (the labels of $S_1 \| S_2$ are then in $\Sigma_1 \cup \Sigma_2 \cup (\Sigma_1 \times \Sigma_2)$).

The *synchronized product* models synchronized parallel composition of programs. Transitions in S_1 and S_2 are either non-synchronizing, case in which they execute independently as in the asynchronous product, or synchronizing, case in which they must execute in parallel.

Definition 5.5 (synchronized product) *Let $\sigma \subseteq (\rightarrow_1 \times \rightarrow_2)$ be a relation between the transitions of S_1 and S_2, which defines the pairs of transitions that are synchronizing. We will denote $\sigma|_{\rightarrow_1}$ and respectively $\sigma|_{\rightarrow_2}$ the projections of this relation on $\rightarrow_1$ and $\rightarrow_2$, i.e. the synchronizing transitions of S_1 and respectively S_2. Let ϵ be a transition label not contained in $\Sigma_1 \cup \Sigma_2$.*

The synchronized product *of S_1 and S_2 according to σ is the LTS*

$$S_1 \otimes_\sigma S_2 = (Q_1 \times Q_2, Q_{0\,1} \times Q_{0\,2}, (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}), \rightarrow)$$

where $\rightarrow$ is the minimal set of transitions defined by the following rules:

1. $\forall (q_1, a, q_1') \in \rightarrow_1$ *such that* $(q_1, a, q_1') \notin \sigma|_{\rightarrow_1}$, *and* $\forall q_2 \in Q_2$, *we have* $(q_1, q_2) \xrightarrow{(a,\epsilon)} (q_1', q_2)$.
2. $\forall (q_2, a, q_2') \in \rightarrow_2$ *such that* $(q_2, a, q_2') \notin \sigma|_{\rightarrow_2}$, *and* $\forall q_1 \in Q_1$, *we have* $(q_1, q_2) \xrightarrow{(\epsilon,a)} (q_1, q_2')$.
3. $\forall ((q_1, a, q_1'), (q_2, b, q_2')) \in \sigma$, *we have that* $(q_1, q_2) \xrightarrow{(a,b)} (q_1', q_2')$.

A variant of synchronized product frequently used in the literature is based on the equality of labels. This is equivalent to considering the following set of synchronizing transitions:

$$\sigma = \left\{((q_1, a, q_1'), (q_2, b, q_2')) \in (\rightarrow_1 \times \rightarrow_2) \quad | \quad a = b\right\}$$

5.3 The timed automata model

Timed automata were defined by Alur et al. in [ACD93, AD94]. Several slightly different versions of the basic model described in [ACD93] have been used in the literature. In the following, we will use timed automata with *urgency*, defined in [BS97, BST98].

Definition 5.6 (timed automaton) *A* timed automaton *is a tuple* $A = (\Sigma, \mathcal{X}, Q, q_0, E)$ *where:*

1. Σ *is a finite set of transition labels.*
2. $\mathcal{X}$ *is a finite set of clocks.*
3. Q *is a finite set of discrete states.*
4. q_0 *is a distinguished state of* Q *called* initial state.
5. E *is a set of transition edges between the states from* Q*, each edge* $e = (q, \zeta, u, a, X, q') \in E$ *having the following components:*
   - $q, q' \in Q$ *are the source and destination states (denoted* $source(e)$ *and* $dest(e)$ *respectively).*
   - ζ *is the guard of the transition (denoted* $guard(e)$*) and it is a conjunction of* atomic conditions *involving clocks from* $\mathcal{X}$*.*

     An atomic condition has one of the following two forms: $x \sim \mathbf{c}$ *or* $x - y \sim \mathbf{c}$ *where* $x, y \in \mathcal{X}$*,* $\sim \in \{<, \leq, >, \geq\}$ *and* $\mathbf{c} \in \mathbb{Z}_+$ *is a constant. We will denote* $\mathcal{CP}(\mathcal{X})$ *the set of conjunctions of atomic conditions over the clocks of* $\mathcal{X}$*.*
   - $u \in \{eager, lazy, delayable\}$ *is an attribute called the* urgency *of the transition (denoted* $urgency(e)$*).*
   - $a \in \Sigma$ *is the label of the transition edge* e *(denoted* $label(e)$*).*
   - $X \subseteq \mathcal{X}$ *is the set of clocks reset during the transition* e *(denoted* $reset(e)$*).*

A semantics for timed automata in terms of labeled transition systems is given in the next paragraph. Informally, timed automata are finite state machines extended with a set of real-valued clocks. Clocks are synchronized, in the sense that they increase at the same rate (extended versions of timed automata, in which this condition is relaxed, were also proposed in the literature; they are discussed in the beginning of §5.5).

A *run* of a timed automaton is a sequence of instantaneous transitions, interleaved with waiting periods in which the automaton resides in a state. Thus, from an operational point of view, while a timed automaton is in a state it has two options: to take a discrete transition (if the transition is *enabled*, i.e. the guard condition holds) or to let some time pass (if the urgency of the enabled transitions allows it). Transitions are executed instantaneously (i.e. the value of the clocks does not increase during the transition) and may reset some clocks to 0.

The notion of *urgency* in the above definition is essential, as it allows a simple modeling of *deadlines*, which appear frequently in real-time models. The urgency attribute of a transition has the following meaning:

- *eager*: the transition does not let time progress. If the automaton is in a state and an *eager* transition is enabled, the automaton may not remain in the state and must take one of the enabled transitions immediately. Note that the transition that is taken may be different from the eager transition in cause.
- *lazy*: the transition lets time progress by whatever amount. If the automaton is in a state and a *lazy* transition is enabled, the automaton may take the transition or may let time pass (if the other enabled transitions allow it too).
- *delayable*: the transition lets time progress up to a limit, beyond which the transition would be disabled. For example, if a delayable transition has a guard $x \leq 2$ and the state is reached with a value of the clock x smaller than 2, time may pass up to the point when $x = 2$. At that point, the transition becomes *eager* and it (or another enabled transition) must be taken immediately.

In contrast to the above definition, the basic timed automata proposed in [ACD93] specify time progress conditions using state invariants. Invariants are boolean conditions on clocks which must hold in the state they refer to. Therefore, in a state, time may progress as long as the invariant remains true.

[BST98] argues that there is an inconvenient in specifying time progress conditions using state invariants: state invariants must continuously hold from the moment the state is entered until the state is exited. With state invariants it is sometimes difficult to model *eager* urgency, i.e. that a transition must be executed as soon as it is enabled.

The notion of urgency originated from that of *deadline* [BS97, BST98]. A *deadline* is a boolean expression involving clocks that is associated with a transition and not with a state as the state invariant. The deadline gives priority of the transition with respect to time progress: while the deadline is false, the transition is not urgent and the time may advance (if the other transitions allow it too). When the deadline is true, the transition is urgent and time can no longer progress until the concerned transition or another enabled transition is fired.

The classes of transition urgency described above correspond to particular cases of deadlines appearing frequently in real-time specifications: $d = false$ for lazy transitions, $d = g$ for *eager* transitions, and $d =$ the upper limit of g for *delayable* transitions (d and g denote respectively the deadline and the guard of the transition).

Semantics of timed automata as labeled transition systems

A semantics is given to timed automata by associating an (infinite) labeled transition system (LTS) G_A to each timed automaton A. This infinite LTS, called the *semantic graph* of the timed automaton, is defined by the following:

1. The nodes of G_A are called *configurations* or *dynamic states* of A. They are pairs $(q, \mathbf{v})$ where $q \in Q$ is a discrete state and $\mathbf{v}$ is a *valuation* of the clocks of the automaton, $\mathbf{v} : \mathcal{X} \to \mathbb{R}_+$. If $s = (q, \mathbf{v})$ is a configuration, we denote by $\mathbf{discrete}(s)$ the discrete state q.
2. The edges of G_A correspond to transitions of A from one configuration to another. There are two kinds of transitions allowed in a state $(q, \mathbf{v})$:

- **Discrete transitions** happen when a transition edge $e = (q, \zeta, u, a, X, q')$ is taken. e is *enabled* in $(q, \mathbf{v})$ iff $\mathbf{v}$ satisfies the condition ζ (also denoted $\mathbf{v} \in \zeta$). When the transition e is taken, the system moves to state $(q', \mathbf{v}')$ where $\mathbf{v}'(x) = \mathbf{v}(x), \forall x \in \mathcal{X} \setminus X$ and $\mathbf{v}'(x) = 0, \forall x \in X$. The transition is denoted by $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')$.
- **Time transitions** happen when an amount $\delta \in \mathbb{R}$, $\delta > 0$ of time elapses without any discrete transition being fired in the meantime. A time transition moves the system from state $(q, \mathbf{v})$ to state $(q, \mathbf{v} + \delta)$ where $\mathbf{v} + \delta$ denotes the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = \mathbf{v}(x) + \delta, \forall x \in \mathcal{X}$. The transition is denoted $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v} + \delta)$. The time transition is enabled iff the following time progress conditions hold:
  (a) $\forall \delta' \in [0, \delta)$, there is no *eager* transition $e = (q, \zeta, u, a, X, q')$ enabled in $(q, \mathbf{v} + \delta')$ (i.e. $u = eager$ and $\mathbf{v} + \delta' \in \zeta$).
  (b) $\forall \delta'$, δ'' such that $0 \leq \delta' < \delta'' \leq \delta$, there is no *delayable* transition $e = (q, \zeta, u, a, X, q')$ enabled in $(q, \mathbf{v}+\delta')$ and disabled in $(q, \mathbf{v}+\delta'')$ (i.e. $u = delayable$ and $\mathbf{v} + \delta' \in \zeta$ and $\mathbf{v} + \delta'' \notin \zeta$).

We considered that G_A contains only the vertices reachable from the initial configuration of the system, which is $(q_0, \mathbf{v})$ with $\mathbf{v}(x) = 0,\ \forall x \in \mathcal{X}$ (also denoted $(q_0, 0)$).

Runs. Canonical representation. Zeno runs

The *runs* of an automaton A are the runs of the semantic graph G_A,regarded as an LTS. A run is therefore an infinite sequence $(q_0, \mathbf{v}_0) \xrightarrow{a_0} (q_1, \mathbf{v}_1) \xrightarrow{a_1} ...$, where the labels $a_0, a_1, ...$ denote either discrete transitions or time transitions.

Two runs which exhibit the same discrete transitions and the same accumulated delays between successive discrete transitions can be considered equivalent. That is to say that two consecutive time steps $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}') \xrightarrow{\delta'} (q, \mathbf{v}'')$ in a run are equivalent to a single time step $(q, \mathbf{v}) \xrightarrow{\delta+\delta'} (q, \mathbf{v}'')$. Moreover, we can consider there is a time transition with delay 0 between any two consecutive discrete transitions of a run.

By the above rules, each run is equivalent to a run of the following form: $\rho = (q_0, \mathbf{v}_0) \xrightarrow{\delta_0} (q_0, \mathbf{v}_0 + \delta_0) \xrightarrow{e_0} (q_1, \mathbf{v}_1) \xrightarrow{\delta_1} (q_1, \mathbf{v}_1 + \delta_1) \xrightarrow{e_1} ...$ in which time transitions and discrete transitions alternate. This is called the *canonical form* of timed automata runs.

A special form of canonical runs appears when the initial form of the run contains an infinite number of transitions among which only a finite number are *discrete* transitions. This implies that there is a point i in the run beyond which all transitions $a_i, a_{i+1}, \dots$ are *time* transitions (with the delays, say, $\delta_i, \delta_{i+1}, \dots$). In this case, the canonical form of the run contains a final transition $\xrightarrow{\delta}$ where $\delta = \sum_{j \geq i} \delta_j$ is the limit of the series of delays (it is possible to have $\delta = \infty$). We denote the canonical form like this: $\rho = (q_0, \mathbf{v}_0) \xrightarrow{\delta_0} (q_0, \mathbf{v}_0 + \delta_0) \xrightarrow{e_0} (q_1, \mathbf{v}_1) \xrightarrow{\delta_1} (q_1, \mathbf{v}_1 + \delta_1) \xrightarrow{e_1} ... \xrightarrow{e_{k-1}} (q_k, \mathbf{v}_k) \xrightarrow{\delta}$.

On the canonical form ρ of a run, we use the following notation to denote the state reached after i time steps and i discrete steps: $\rho(i) = (q_i, \mathbf{v}_i)$.

A configuration $(q, \mathbf{v})$ is *reachable* if there is a run of the automaton, starting in the initial state $(q_0, 0)$ and ending in $(q, \mathbf{v})$.

A run of the canonical form shown above is called *zeno* if it is infinite and the total elapsed time along the run is finite, i.e. $\sum_{i \geq 0} \delta_i \in \mathbb{R}$. The interpretation is that an infinite number of discrete transitions (which normally model *actions* in the specified system) is executed in a

finite amount of time along a *zeno* run. This usually corresponds to a erroneous or incomplete specification.

Timed automata composition

As for LTS, it is sometimes possible to describe a concurrent timed system using a set of timed automata modeling concurrent components, and a *composition* rule for constructing the global TA of the system. [Bor98] proposed several composition operators for TA with urgency.

We present here a variant of synchronized composition operator, which corresponds to the synchronized execution of some discrete transitions and the interleaved execution of other discrete transitions. The pairs of synchronizing transitions are given by a binary relation σ. Synchronous passage of time in all components is assumed.

Definition 5.7 (sychronized product) *Let $A = (\Sigma, \mathcal{X}, Q, q_0, E)$ and $A' = (\Sigma', \mathcal{X}', Q', q_0', E')$ be two TA with urgency. Also, let $\sigma \subseteq E \times E'$ be a binary relation between transitions from A and A' which denotes which pairs of transitions are synchronizing.*

The synchronized composition of A and A' is $A \otimes_\sigma A' = ((\Sigma \cup \{\epsilon\}) \times (\Sigma' \cup \{\epsilon\}), \mathcal{X} \cup \mathcal{X}', Q \times Q', (q_0, q_0'), T)$, where T is the minimal set of transition edges defined by the following rules:

1. $\forall e = (q_1, \zeta, u, a, X, q_2) \in E$ *such that* $\nexists e' \in E'.\ (e, e') \in \sigma$, *and* $\forall q' \in Q'$, $((q_1, q'), \zeta, u, (a, \epsilon), X, (q_2, q')) \in T$.
2. $\forall e' = (q_1', \zeta', u', a', X', q_2') \in E'$ *such that* $\nexists e \in E.\ (e, e') \in \sigma$, *and* $\forall q \in Q$, $((q, q_1'), \zeta', u', (\epsilon, a'), X', (q, q_2')) \in T$.
3. $\forall e = (q_1, \zeta, u, a, X, q_2) \in E$ *and* $\forall e' = (q_1', \zeta', u', a', X', q_2') \in E'$ *such that* $(e, e') \in \sigma$, *then* $((q_1, q_1'), \zeta \wedge \zeta', max(u, u'), (a, a'), X \cup X', (q_2, q_2')) \in T$

The composition operator for urgencies, max is defined as the maximum with respect to the following order relation: $lazy < delayable < eager$. The label ϵ denotes the fact that one component is not taking any transition, in interleaved transitions.

The TA composition operator defined above is the simplest form of composition, and corresponds to the synchronized composition of the semantic graphs (LTSs) of the two automata. The operator defined above is equivalent to the AND composition operator defined in [Bor98], provided the following restriction holds: if two synchronized transitions have urgencies *delayable* and respectively *lazy*, then the guard of the *lazy* transition must be $\zeta = true$. A more complicated AND composition rule is defined in [Bor98] for the cases when the above condition does not hold. For more details, the reader is referred to [Bor98].

The composition of timed automata is not used explicitly in the remaining of this thesis, as we discuss the semantics of time in SDL directly at the level of the semantic graph of an entire SDL system. However, the rules by which describe the transitions (with their characteristics, like guards and urgencies) of that graph correspond to this definition of composition, applied to SDL agents. The composition operator would be explicitly necessary if a complete compositional description of a timed automata-based semantics for SDL was aimed.

Example of timed automata specification

We illustrate the timed automata model introduced previously, using a slightly modified version of a classical example first introduced in [Alu91]. It models the controller of a railroad crossing gate system. The controller interacts with two elements of its environment: a train proximity

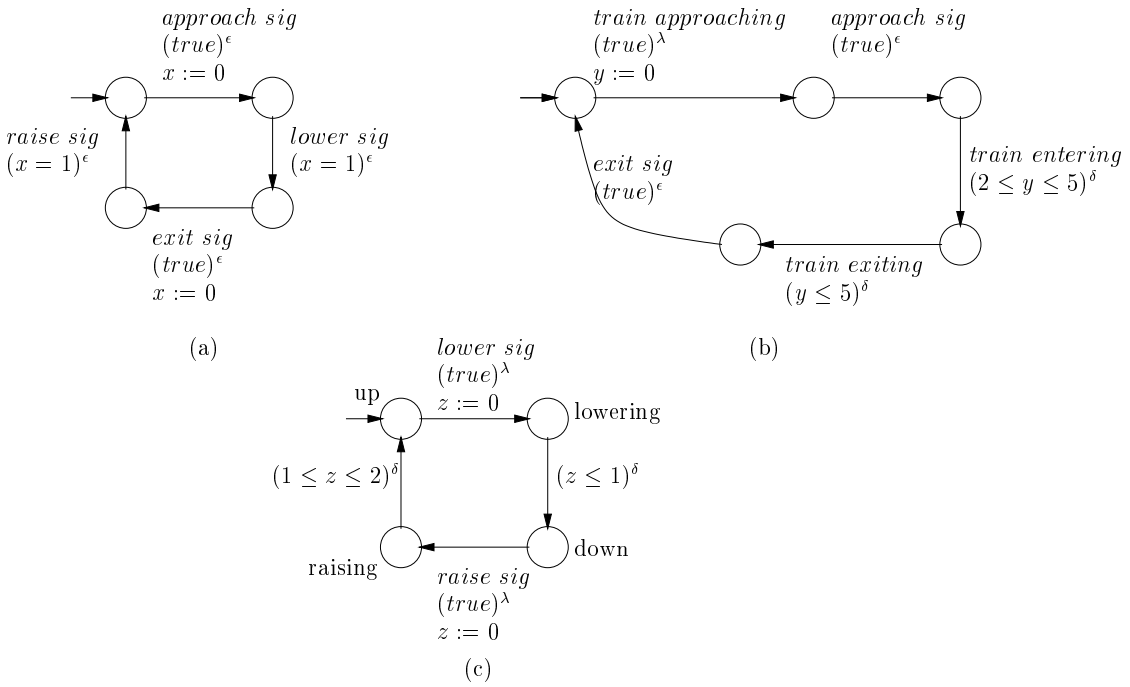


Figure 5.1: The railroad gate system modeled with timed automata.

sensor, and the actuators of the physical gate. Its behavior is quite simple and deterministic: each time the proximity sensor signals the approach of a train, the controller waits for one time unit then begins to lower the gate. Then, when the sensor signals the *exit* of the train from the zone of the gate, the controller waits for one time unit after which it begins to raise the gate.

The behavior of the controller is modeled in Fig. 5.1-a. In the figure, states are represented as circles, the initial state being marked with a dangling incoming arrow. States are annotated with a name, when that is significant. Transitions are represented as arrows between the source state and the destination state, annotated with their label, clock guard, urgency (we use λ,δ and ϵ exponents to denote respectively *lazy*, *delayable* and *eager* urgency, like in [BS97]), and clock resets. Clocks are denoted with characters from the end of the roman alphabet.

All the transitions of the automaton modeling the controller are marked as eager, because they correspond to actions executed by the controller as soon as they are possible. The waiting times are modeled by testing the value of clock x.

For assessing the timing of this system, the environment of the gate controller must also be modeled. We model the components of the environment through two other automata, which synchronize with the controller automaton (transitions with identical labels are synchronizing).

Although the complete behavior of the environment is not completely deterministic, some information about it may still be available. For example, we consider that an approaching train is detected by the sensor at least 2 time units before it enters the gate, and that it exits the gate at most 5 time units after it has been first detected. Moreover, the proximity sensor is considered to transmit the *approaching* signal to the controller as soon as the train is detected, and the *exit* signal as soon as the train exits the gate. The automaton in Fig. 5.1-b represents the train and the proximity sensor together. Transitions concerning the train (*train approaching*, *entering*, *exiting*) are *lazy* or *delayable*, modeling the non-determinism of the train. Transitions

which model the sending of signals to the controller are *eager* to model the immediate reaction of the sensor.

The behavior of the gate, shown in Fig. 5.1-c, is as follows: it takes between 0 and 1 time units for the gate to go down, and between 1 and 2 time units to go up. Transitions which start lowering and raising the gate are modeled as lazy, to capture the fact that they are triggered by synchronizing with the controller.

5.4 Analysis techniques and decidable problems

An interesting problem concerning a timed automaton A is whether a particular configuration $(q, \mathbf{v})$ is reachable from the initial state $(q_0, 0)$. This is called the *reachability* problem. The verification of many properties of TA, such as invariance or other safety properties, can be reduced to reachability.

[ACD93] provides a solution for the reachability problem, using an abstraction technique which allows to build a finite graph, which preserves reachability, from the potentially infinite semantic graph G_A. The abstract graph, called *region graph*, is defined below.

The region graph

In what follows, it will be useful to give a geometrical representation to the clock space of a TA: in a configuration $(q, \mathbf{v})$ of an automaton A, $\mathbf{v}$ is point in the space $\mathbb{R}^{|\mathcal{X}|}$. A conjunction of atomic conditions $\zeta \in \mathcal{CP}(\mathcal{X})$ defines a convex polyhedron in $\mathbb{R}^{|\mathcal{X}|}$; the polyhedron can be identified with the condition ζ. A disjunction of conditions ζ defines a non-convex polyhedron in $\mathbb{R}^{|\mathcal{X}|}$; we will denote $\mathcal{NCP}(\mathcal{X})$ the class of non-convex polyhedra on $\mathcal{X}$.

The following definition of the region graph is based on that from [Tri98], with some small corrections (namely, the addition of condition no. 3 in the definition below). The construction of the region graph is based on the observation that the transition guards of a TA, involving only conditions such as $x \sim \mathbf{c}$ or $x - y \sim \mathbf{c}$ (see Def. 5.6), cannot distinguish between two valuations $\mathbf{v}$ and $\mathbf{v}'$ if the integral part of all clocks is the same and the order of the fractional parts is the same. Moreover, for each TA there is a maximal constant c with which a clock or a clock difference is compared, and the transition guards cannot distinguish between values of clocks or differences exceeding c.

Formally, the construction of the region graph is based on the definition of the *region equivalence* relation: two valuations $\mathbf{v}$ and $\mathbf{v}'$ are region equivalent with respect to the maximal constant c ($\mathbf{v} \simeq_c \mathbf{v}'$) iff:

1. $\forall x \in \mathcal{X}.\ \lfloor \mathbf{v}(x) \rfloor = \lfloor \mathbf{v}'(x) \rfloor$ or $\lfloor \mathbf{v}(x) \rfloor > c$ and $\lfloor \mathbf{v}'(x) \rfloor > c$.
2. $\forall x, y \in \mathcal{X}.\ \lfloor \mathbf{v}(x) - \mathbf{v}(y) \rfloor = \lfloor \mathbf{v}'(x) - \mathbf{v}'(y) \rfloor$ or $|\mathbf{v}(x) - \mathbf{v}(y)| > c$ and $|\mathbf{v}'(x) - \mathbf{v}'(y)| > c$.
3. $\forall x \in \mathcal{X}.\{\mathbf{v}(x)\} = 0 \Leftrightarrow \{\mathbf{v}'(x)\} = 0$.

(where $\{r\}$ and $\lfloor r \rfloor$ denote respectively the fractional and the integer part of a real number r).

Fig. 5.2 shows the region equivalence classes for a 2-clock space, with a maximal constant $c = 2$ (points, lines and grayed zones represent equivalence classes).

The region equivalence on clock valuations induces an equivalence relation between the nodes of G_A: $(q, \mathbf{v}) \simeq_c (q', \mathbf{v}')$ iff $q = q'$ and $\mathbf{v} \simeq_c \mathbf{v}'$. [ACD93] proves that $\simeq_c$ is a strong time

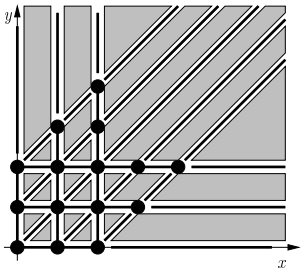


Figure 5.2: The region equivalence classes for two clocks and $c = 2$

abstracting bisimulation of A[1], i.e. if concrete time values are abstracted away on the time transitions of G_A, region equivalence is a *strong equivalence* on G_A in the LTS sense (§5.2).

The region graph is defined as the quotient of G_A with respect to $\simeq_c$. It preserves *reachability* in the sense that a configuration $(q, \mathbf{v})$ is reachable in A if and only if its region equivalence class $\widehat{(q, \mathbf{v})}$ is reachable in $G_A/_{\simeq_c}$. As the region graph is finite ([ACD93] provides an upper bound for the size of the graph), and there is an effective procedure for representing and computing the region graph of a TA, it follows that the reachability problem is decidable for TA.

Other forms of the reachability property, such as: "starting from a state $(q, \mathbf{v})$, can the automaton A reach a discrete state q'?" can also be solved using the region graph. Moreover, the region graph preserves more complex classes of properties, such as linear-time properties or branching time properties expressed in some temporal logics. A survey can be found in [Tri98].

The simulation graph

For verification problems involving only reachability or linear properties, there are more efficient analysis methods than the region graph mentioned above. In this section we describe the *simulation graph*, that is used in a later chapter for the verification of temporal properties of SDL specifications.

The *simulation graph* of A has vertices of the form (q, S), where $q \in Q$ is a discrete state and $S \in \mathcal{NCP}(\mathcal{X})$ is a polyhedron that we will call *zone*. The following operations are defined on (q, S) pairs:

$$\begin{aligned} \mathsf{time\text{-}succ}((q,S)) &= (q, \{\mathbf{v}' \mid \exists \mathbf{v} \in S,\ \delta \in \mathbb{R}.\ (q,\mathbf{v}) \xrightarrow{\delta} (q,\mathbf{v}')\}) \\ \mathsf{disc\text{-}succ}(e,(q,S)) &= (q', \{\mathbf{v}' \mid \exists \mathbf{v} \in S.\ (q,\mathbf{v}) \xrightarrow{e} (q',\mathbf{v}')\}) \end{aligned}$$

where e is an edge between q and q'.

It can be proved that if S is a zone from $\mathcal{NCP}(\mathcal{X})$, $\mathsf{time\text{-}succ}((q,S))$ and $\mathsf{disc\text{-}succ}(e,(q,S))$ also yield zones from $\mathcal{NCP}(\mathcal{X})$[2]. The simulation graph of the automaton A is the smallest graph $SG(A)$ such that:

1. $\mathsf{time\text{-}succ}((q_0, 0))$ is a node of $SG(A)$
2. for every node (q, S) of $SG(A)$ and every discrete transition edge e from q to q', if $(q', S') = \mathsf{time\text{-}succ}(\mathsf{disc\text{-}succ}(e,(q,S)))$ and $S' \neq \emptyset$ then (q', S') is also a node of $SG(A)$ and $(q, S) \xrightarrow{e} (q', S')$ is an edge of $SG(A)$.

[1] The results in [ACD93] refer to the basic TA model. However, they can be easily extended to TA with urgency.

[2] For basic TA (without urgency), convexity is also preserved, and therefore all zones of the simulation graph are from $\mathcal{CP}(\mathcal{X})$. See [Tri98].

We note that a zone S can be decomposed in a finite union of *regions* (see previous section). In consequence, the simulation graph is always finite. It can further be proven that the simulation graph preserves reachability and linear properties (i.e. every run of A is contained in a path from $SG(A)$). For further details, proofs and examples, the reader is referred to [Tri98].

5.5 Discussion

Decidability limits of the timed automata model

A number of extensions of the basic timed automata model have been studied. They try to overcome practical limitations of the TA, e.g. by generalizing the laws of variation for clocks or the types of conditions that can be included in guards. We survey here some of these lines of research.

Automata with integrators allow to measure accumulated delays by using clocks which can be stopped and restarted at the same value (called integrators or stopwatches). They are useful for example to model preemptive multitasking systems. However, the reachability problem for timed automata with integrators is undecidable [Cer92, HKPV98].

There have been several attempts to define *restricted* variants of integrator automata for which reachability is decidable [KPSY93, BER94, ACH93]. The restrictions are quite important: for example, [KPSY93, ACH93] constrain the integrators to be neither reset not tested by the automaton, except in a final transition that may be triggered only once.

Multirate automata [ACH+95] define clocks that may vary at different relative speeds. They are useful for modeling distributed systems with drifting clocks. The reachability problem for multirate automata is decidable with the condition that clocks are not compared between them but only with constants (i.e. no $x - y \sim \mathbf{c}$ conditions allowed in guards) [HKPV98, ACH+95].

In general, TA and all their extensions are restrictions of a more general model, *hybrid automata* [ACHH93, NOSY93]. A hybrid automaton models a *hybrid system* [MMP91, NSY91], which combines discrete and continuous components. Hybrid automata are state-transition systems in which the state has a discrete part and a continuous part. The continuous part is formed of real-valued variables which vary in time according to a law, which can be very general (e.g. a differential equation). TA are hybrid systems in which all continuous variables x (*clocks*) vary by the equation $\dot{x} = 1$. Hybrid automata have been extensively studied in the past decade, with results ranging from the identification of decidable restrictions [HKPV98, Hen96] to various verification methods applicable on restricted models [ACH+95, OSY94].

All the models mentioned above are usually decidable only under strong restrictions (for a synthesis of decidability results, see [HKPV98] and the work cited therein). *Semi-decision* procedures can sometimes be developed for undecidable models. Such procedures are important from a practical point of view, in situations where timed automata are not expressive enough and only a generalized model can capture the behavior of a system. Nevertheless, these semi-decision procedures are usually complex and difficult to apply to large models. Thus, timed automata give in a way a *complexity limit* up to which "general" timed models are decidable, and the available analysis methods are simple enough to be applied in a framework based on SDL. Therefore, in our work we have restricted to basic timed automata and equivalent SDL models.

Automata vs. other models for timed behavior

Much of the research on timed models has initially concentrated on other kinds of formalisms, such as Petri Nets or process algebras. In the domain of Petri Nets, we mention the early models of Timed Petri Nets [Sif77, Ram74], the Time Petri Nets [MF76], as well as the many other variations defined subsequently (for a recent comparison between different models, see [Boy01]).

Petri Nets present some advantages from the modeling point of view, being able to capture more naturally different types of synchronization and composition. However, recent research has concentrated more on basic automata models, for which more evolved constructs for specifying timing constraints, urgency, etc. have been developed. From the point of view of the expressivity of the models, there are several results showing the equivalence between classes of time Petri Nets and classes of timed automata (see the survey in [Boy01]). From the point of view of analysis, both models use essentially the same techniques, based on the construction of a symbolic state space, and on inequality systems for representing time information. We note however that the application of these techniques in verification tools is more advanced on the side of automata-based models, with tools such as Kronos [Yov97, DOTY95], Uppaal [LPY97, BLL+96], HyTech [HHWT97] and IF [BFG+99, Boz99].

The situation is similar on the side of timed extensions of process algebras, where analysis techniques typically work by mapping the algebraic model to some automata model similar to timed automata (see for example [Nic92]).

Part II

Language Extensions, Validation Techniques and Tools

Chapter 6

SDL extensions for timed behavior description

In this chapter we examine a series of extensions to SDL which improve the capability of the language to handle the specification of time-related information at an abstract level. The technique for specifying timing information introduced here uses the primitive mechanisms from timed automata: clocks, conditions on clocks and urgencies. The semantics of time in SDL is adapted to suit the usage of these mechanisms.

We begin the chapter with an overview of the problems encountered when specifying time-related behavior in SDL. The issues discussed here have been pointed out in our previous work [BGK+00, BGM+01], and an outline is given in §3.5.

We continue in §6.2 by introducing the extensions to the SDL language. The constructs proposed here are inspired from primitives used in timed automata. For this reason, they are rather low level, and current efforts go towards distilling a set of higher level primitives (semantically based on those introduced here) to be proposed for standardization [BGM+01].

In §6.3 we examine the impact of the proposed extensions on the formal semantics of SDL described in the Annex F of Z.100 [IT99c] . The precise semantics of the extensions as well as the new semantics of time is expressed in a clear and formal way using Abstract State Machine (ASM) specifications.

As current simulation and verification tools based on SDL do not use the standard ASM semantics from [IT99c], in §6.4 we discuss how the extensions defined previously may be integrated in the LTS-based semantics used by most tools. The result is an LTS that has all the characteristics of the semantic graph of a timed automaton: the states contain a discrete part (referring to the agents, states, variables, etc. of the SDL system) and a part referring to clocks, the transitions are either *discrete* transitions or *time* transitions. The simulation and verification tool described in Chapter 8 uses the LTS-based semantics of the extended SDL defined here, in connection with analysis techniques originating from timed automata.

We close the chapter with a discussion of the gains brought by the extensions, and of related work that can be found in the literature.

6.1 Overview of problems

We discussed previously (§3.5) the dual nature of SDL which is both a *specification* formalism and a *programming* formalism, and we have outlined the fact that SDL gives precedence

to programming constructs rather than high-level specification. [BGK+00, BGM+01] provide suggestions for language improvements on both sides. In this thesis we concentrate on the *specification* side, as that is more critical for the validation of real-time system specifications.

6.1.1 Classification of problems and solutions

The use of SDL as a real-time specification formalism leads to two types of problems: *expressivity* problems and *usability* problems.

1. *Expressivity* problems are represented by the impossibility to capture in SDL meaningful (timing) information about a system, like the execution time boundaries of a piece of SDL code. This kind of problems is due to the lack of appropriate language constructs for expressing such information.

2. *Usability* problems are caused by the practical or theoretical impossibility to use SDL models for some specific system engineering task, like simulation or property verification. Usability problem are frequently due to the definition of the SDL *semantics*. For example, with the present definition of the SDL semantics, there are currently no analysis methods for deciding the reachability problem on SDL models in the general case.

The solutions for the two types of problems are different: *expressivity* problems require the addition of new constructs to the language, whereas *usability* problems require the modification of the language semantics. In the definition of new constructs, care must be taken as to the coherence between new constructs and existing ones: no overlapping and no hidden dependencies should exist. The modification of the language semantics is however more problematic, as the same semantics normally has to serve several purposes (e.g. code generation, simulation, verification, performance analysis, etc.), which may impose contradictory demands.

We take the example of code generation versus formal verification. The semantics of SDL [IT99c] is more suitable for code generation than for simulation and verification: [IT99c] maintains that each action takes an indeterminate time to execute, and that a process stays an indeterminate amount of time in a certain state before taking the next fireable transition. This notion of time that is external and unrelated to the SDL system is practical for code generation in the sense that actual implementations of the system conform to it. However, for simulation and verification, this semantics of time is impractical: timer extents do not have any significance except that of lower time bounds, and any timer that gets in a queue may stay there for an indeterminate amount of time.

Any rigorous attempt to construct the semantic graph (LTS) of an SDL system (which is the starting point for simulation and verification) must account for all combinations of execution times, timer expirations and timer consumptions, causing an explosion of the state space. Moreover, few temporal properties may be ensured using the hypotheses stated by the standard semantics. This causes a *usability* problem.

In practice, simulation and verification tools make simplifying assumptions on execution and idle times. The usual convention is that actions take 0 time to execute, and any action that can be executed is executed immediately. This option is justified by the fact that it generates the highest degree of determinism, thus reducing the state space by an important factor and rendering SDL specifications analyzable.

The two alternative definitions of the SDL semantics mentioned above are mutually exclusive and equally justified: one by the needs of code generators, one by the needs of simulators and

verification tools. We argue that this dichotomy cannot be surpassed by a single SDL semantics. A solution is to adopt multiple semantic *profiles* of SDL, which would correspond to different usages of SDL models: code generation, simulation, performance analysis, model checking, test generation etc. A semantic profile would define a semantics that is particularly suitable for a certain type of manipulation. We consider however that the definition of a theoretical basis for defining profiles is outside the scope of this thesis, and we do not explore this issue in more detail.

6.1.2 Expressivity problems

In this section we outline some problems that may be encountered when expressing abstract timing information in an SDL specification. The extensions described in later sections go towards solving the problems enumerated here.

Assumptions on execution times

The abstract specification of a real-time system may involve the specification of information about the execution times of certain actions. Such descriptive information is meaningful in simulation and verification, as the well functioning of the system may depend on it.

Currently, in order to introduce assumptions on execution times the modeler is forced to use imperative constructs such as timers. While this does not entirely solve the problem (e.g. maximal execution times cannot be expressed), it also implies a style of specification which is incompatible with some uses of the SDL model (e.g. code generation).

Execution times normally depend on the deployment of the SDL system. There are however cases in which execution times are meaningful from a qualitative point of view (e.g. for calibrating system timers), and thus appearing in earlier phases of system specification.

Some earlier approaches for the specification of execution times in SDL models exist [Rou98, DHHMC95a]. The *Object*GEODE Simulator [Rou98] uses a syntactic extension by which one can associate an execution time interval to an action, and a probability distribution in this interval. [DHHMC95a] uses a more elaborate approach in which execution times are dynamically calculated with the help of queuing machines, so that they are depending on the amount of work and on the charge of the system. However, both approaches target performance evaluation and lack precise semantic definition and a unified mechanism for expressing other timing assumptions (e.g. communication times or the timing of events, see next paragraphs).

Assumptions on timing of events

In open specifications communicating with the environment, the timing of events coming from the environment is an important factor for the behavior of the system. We argue that information of the timing of these events should be included in the SDL specification. Moreover, the development of a real-time system usually comports several preliminary stages in which abstract and incomplete descriptions are produced. In order to validate these early designs, the timing of events occurring in incompletely specified components has to be described within the SDL model.

To preserve the clarity of the language, the extensions for expressing timing assumptions have to be based on a simple primitive mechanism capable of expressing many types of timing assumptions (e.g. event period and jitter, timed inter-event synchrony, etc), instead of enumer-

ating different extensions for all the types of assumptions. This concerns also the expression of execution times, and the other timing information discussed in this section.

Assumptions on channel behavior

SDL defines channels as reliable means for transporting messages: a channel never looses messages. Additionally, a channel may either be non-delayable (i.e. messages arrive instantaneously at the other end) or with non-specified delays (but keeping the order of the conveyed messages).

These attributes are insufficient for characterizing real communication channels. For example, SDL is used to describe flow control protocols such as the alternating bit protocol from the OSI stack. Such protocols are built upon the assumption that channels are unreliable, and it is their mission to make them reliable through software. If the assumptions on channels cannot be marked in SDL, the resulted description of the protocol cannot be used in simulation or verification: the tools will never cover the behavior parts that handle signal loss.

Currently, lossy or delaying channels can be modeled only by explicitly describing the behavior of the channel (e.g. using an SDL process). This approach has several drawbacks:

- once the behavior of the channel is specified, all messages will arrive at destination with a wrong **sender** PId.
- the channel description must be replicated over and over again for every lossy channel in the system (note that a generic process type cannot be used, because the channel description depends on the types of the conveyed signals, which differ from channel to channel).
- dynamic creation of timers is needed in order to transport an indefinite number of messages at once on a delayable channel.

A solution is to let the modeler describe the behavior of the channel through a set of attributes such as loss probability and upper and lower time bounds for transmission delays. More complicated solutions which take into account the type and size of a message can also be imagined.

6.1.3 Usability problems

As described previously, usability problems consist in the difficulty to use the standard semantics of an SDL model for a specific engineering task. In this work we are concerned with model-based validation tasks: simulation, formal verification (model checking). These tasks require the construction of the semantic graph (LTS) corresponding to the behavior of an SDL model. There are two major difficulties in building the graph with the standard semantics: the lack of control over time progress and the lack of an appropriate notion of atomicity.

Control over time progress

This problem was mentioned as an example in §6.1.1. It refers to the fact that using the rules for time passage prescribed by the SDL semantics [IT99c], the semantic LTS of an SDL model will contain many unrealistic execution scenarios. The result is both a state explosion phenomenon and the impossibility to guarantee elementary timing properties.

In order to be usable in simulation or verification, the semantics of an SDL model must prescribe some level of control over the progress of time. Existing simulation tools do this, by

assuming that actions take 0 time to execute, and that time never progresses while the system has something to execute.

These means of controlling the time progress in simulation are limited. There are cases when the user needs to control the simulation time in more flexible ways:

- to specify that in a certain state, an unlimited amount of time may pass, even though the system has something to execute,

- to specify that in a state, a bounded amount of time may pass regardless of whether there is something to execute or not. In this case, there is a number of consequent problems concerning the specification of the amount of time (fixed or with lower and upper bounds; specified statically or dynamically).

Atomicity of transition elements

Z.100 [IT99b] prescribes that the agents composing a system or a block are executed in parallel. However, simulation or verification techniques are based on the LTS associated to an SDL model, which can only be built by assuming a certain degree of atomicity. The formal semantics of SDL [IT99c] is equivalent to an interleaving model at the level of SDL actions (§3.3.3). However, if execution times are associated to individual actions, the validity of this interleaving model has to be checked, since the execution of truly parallel actions should not sum up execution times, while the execution of interleaved actions (e.g. in process sub-agents) should do it.

6.2 Extensions for representing timing information

In this section we introduce some SDL language extensions which allow to express descriptive timing information characterized as problematic in the previous section. The syntax and informal semantics for these extensions are described in the following paragraphs. As a general rule, we describe the *abstract syntax* of the extensions using the BNF-like formalism from Z.100 [IT99b]. The BNF rules described here either replace the productions for existing non-terminals, or specify newly introduced non-terminals.

The definition of the concrete syntax is less formal. As a general rule, we use formal SDL comments (see the **comment** keyword of SDL [IT99b]) for introducing annotations on model entities. However, the syntax proposed here is not the most suitable for additions to the standard, and is provided only to clarify the definition of the extensions.

The semantics of the extensions is explained informally in this section. The impact of these extensions on the formal semantics of SDL is discussed later on in §6.3.

6.2.1 Clocks, guards and transition urgency

We introduce one basic mechanism which can serve for describing many forms of timing information: the clock. Like in timed automata (Chapter 5), clocks can be used to *measure* and *constrain* time passage. A specification may use several clocks, which all progress at the same rate. To preserve the encapsulation principle, each clock must belong to an SDL agent. Only the owner agent of a clock, and its sub-agents may refer to the value of the clock or perform operations on it.

Clock operations

From the point of view of the SDL type system, a clock is a value of the type `Clock`. Clocks can be declared statically, like in:

```
dcl c Clock;
```

or created dynamically using the `mkClock()` operator like in:

```
dcl c object Clock;    /*reference type*/
...
c := mkClock();
```

The `resetClock` construct is used to reset a clock to 0:

```
resetClock(c);
```

The assignment between two `Clock` variables is also allowed. This operation is not defined in timed automata, but it does not interfere with the analysis methods and the decidability of the TA model.

We impose several restrictions to the use of variables of type `Clock`, in order to preserve the applicability of the analysis methods existing for timed automata to SDL. A first restriction is that a clock variable may not be passed as parameter to a procedure, in an agent creation or in an output. Passing the value of a clock as parameter would equate with a *stop clock* operation, which is beyond the scope of timed automata. For compatibility with timed automata, a second restriction is that expressions involving `Clock` variables are allowed only inside transition guards or continuous signals (SDL **provided** clause).

The operators defined below may be used for building expressions involving `Clock` variables. The following set of operators can be used for comparing a clock with an integer value:

```
"<"  : Clock,Integer -> Boolean
"<=" : Clock,Integer -> Boolean
"="  : Clock,Integer -> Boolean
">"  : Clock,Integer -> Boolean
">=" : Clock,Integer -> Boolean
```

The difference of two clocks yields a value of the predefined type `DifClock`. `DifClock` is by definition not compatible with any other type in the SDL type system, so `DifClock` values cannot be converted into reals, for example. The purpose of this constraint is to forbid statically the use of clocks in expressions different from those defined in timed automata.

```
"-" : Clock,Clock -> DifClock
```

The operators predefined for `DifClock` values are:

```
"<"  : DifClock,Integer -> Boolean
"<=" : DifClock,Integer -> Boolean
"="  : DifClock,Integer -> Boolean
">"  : DifClock,Integer -> Boolean
">=" : DifClock,Integer -> Boolean
```

Guards and urgency

In order to specify the precise timing of events, we need a mechanism able to link the moment when a transition is fired to the values of clocks. This is done, like in timed automata, by using transition guards to constrain transition firing and urgencies to constrain time passage.

A transition guard or continuous signal (**provided** clause) can use comparisons of clock values to constrain the moment the transition is fired. The guard must be the conjunction of two parts $c_1 \wedge c_2$ where c_1 is a (possibly void) condition not involving clocks, and c_2 is a (possibly void) *conjunction* of boolean terms obtained from clock comparisons and clock difference comparisons.

Like in timed automata (§5), we define three classes of transition urgency: **lazy**, **delayable**, **eager**. In the abstract syntax, an *urgency* attribute which can take one of the above values is attached to each type of transition clause (*Input-node*, *Continuous-signal*, *Spontaneous-transition* non-terminals):

Input-node :: *Transition-urgency*
[**priority**]
Signal-identifier
[*Variable-identifier*]*
[*Provided-expression*]
[*On-exception*]
Transition

Spontaneous-transition :: *Transition-urgency*
[*On-exception*]
[*Provided-expression*]
Transition

Continuous-signal :: *Transition-urgency*
Continuous-expression
[*Priority-name*]
Transition

Transition-urgency = **lazy** | **delayable** | **eager**

In concrete syntax, urgency is represented with a formal comment (**comment**) attached to the transition clause, containing the urgency attribute. Transitions not specifying the urgency attribute have *eager* urgency by default. This choice differs from the standard semantics of SDL (which is equivalent to considering all transitions *lazy*) but is justified in simulation and verification, as it reduces the size of the state space and leads to more realistic scenarios.

6.2.2 Action execution durations

In the TA model, transitions execute in 0 time. For this reason, in the semantics associated to the extended SDL, we consider that SDL actions also execute in 0 time. However, time consuming actions may be represented explicitly using clocks and urgencies: an additional state represents the time consuming action, and a *delayable* transition exiting this state represents the ending of the execution (time). The actual action may be executed (in 0 time) either upon entering or exiting this state. We introduce an extension for annotating time consuming actions, so that the representation described above is obtained by an implicit transformation.

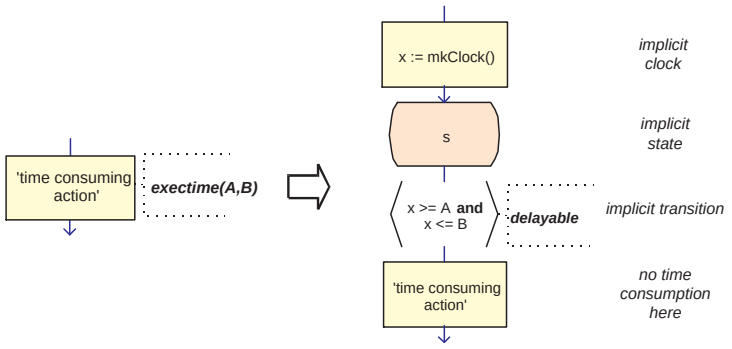


Figure 6.1: Implicit transformation of time consuming actions

Syntax

Graph-node :: (*Task-node*
| *Output-node*
| *Create-request-node*
| *Call-node*
| *Compound-node*
| *Set-node*
| *Reset-node*)
Exec-min-delay
Exec-max-delay
[*On-exception*]

Exec-min-delay = *Nat*

Exec-max-delay = *Nat*

In abstract syntax, two integer attributes (*Exec-min-delay* and *Exec-max-delay*) representing the lower and upper limits of execution time are associated to each action (*Graph-node* non-terminal). In concrete syntax, a time consuming action is annotated with a formal **comment** containing the string `"exectime(A,B)"`, where `A` and `B` are integer constants denoting the lower and upper bounds of the execution time.

The extension is similar to that proposed in [Rou98]. If the execution time of an action is not specified explicitly, the default time limits are both equal to 0. This choice is different from the standard semantics of SDL, but is justified in simulation and verification as it reduces the size of the state space and leads to more realistic scenarios.

Semantics

A time consuming action introduces an implicit state in the enclosing agent. When the system reaches the action, it stays blocked in that implicit state for an amount of time $\delta \in [A, B]$, where A and B are the execution time limits specified for the action. Time consuming actions are shorthand notation for (and translated implicitly into) a 0-time model, as shown in Fig. 6.1.

6.2.3 Channel behavior specification

Standard SDL channels never lose nor distort messages, and the time necessary to transfer a signal can be controlled using the **nodelay** attribute: 0 time if the channel is **nodelay**, indeterminate time otherwise. We introduce here two extensions which allow to specify:

- the possibility of signal loss, with a certain probability,
- the minimal and maximal delays for signal transfer.

These extensions cover the basic modeling needs for a large class of systems. However, if the properties of a channel are more complex (e.g. signal distorsion, loss probability depending on the length of signal, etc.) they have to be modeled explicitly, using for example an SDL process.

Syntax

Channel-definition :: *Channel-name*
[**nodelay**]
Channel-loss-probability
Channel-delay-kind
Channel-min-delay
Channel-max-delay
*Channel-path-**set***

Channel-delay-kind = **delay** | **pipeline**

Channel-min-delay = *Nat*

Channel-max-delay = *Nat*

Channel-loss-probability = *Literal*

In abstract syntax, a signal loss probability (*Channel-loss-probability* – real constant in $[0, 1]$) is associated to a channel specification (*Channel-definition* non-terminal). Two integer attributes (*Channel-min-delay* and *Channel-max-delay*) representing the lower and upper limits of signal transmission delay, and an attribute (*Channel-delay-kind*) taking the value **delay** or **pipeline**, are also associated to each channel specification. The meaning of the delay depends on the attribute **delay** or **pipeline**, as explained in the next section.

In concrete syntax, lossy channels are annotated with a formal **comment** containing `"lossy(p)"`, where `p` is the signal loss probability. Delaying channels are annotated with a formal **comment** containing either `"delay(A,B)"` or `"pipeline(A,B)"`, where `A` and `B` are integer constants denoting the lower and upper bounds of signal communication delay. At most one delay and one loss specification can be mixed in the same **comment** and refer to the same channel.

If the loss probability is not specified explicitly for a channel, the default value is 0. If the transmission delay is not specified explicitly for a channel, the default value is `pipeline(0,0)`. Channels marked with the standard SDL attribute **nodelay** must have the minimal and maximal transmission delays equal to 0.

Semantics

There is a signal queue for each valid direction of every channel instance of an SDL specification. The queue holds the signals that have been sent at one end of the channel, and not yet received

at the other end. Thus, when a signal is transferred between two agents it may pass through several channel queues before arriving at the destination. Each queue preserves the order of the conveyed signals.

Signals sent through a channel marked `lossy` may not arrive at the receiving end of the channel. In that case, no observable event will occur at the receiving end of the channel, and the loss of a signal does not influence the transmission of other signals conveyed by the same channel. The loss probability is informative, in the sense that if `p` is strictly between 0 and 1, for every transferred signal the two alternatives of losing or not losing it are valid behaviors.

A signal sent through a channel marked with `"pipeline(A,B)"` at a moment T arrives at the receiving end at a moment between $T+A$ and $T+B$. The arrival time is further constrained by the fact that the order of the signals is preserved, so a signal may not arrive at the end of a channel before the signal preceding it in the channel queue.

A signal sent through a channel marked with `"delay(A,B)"` at a moment T arrives at the receiving end at a moment between $T'+A$ and $T'+B$, where T' is the maximum between T and the arrival time of the previous signal transferred through the channel.

The two types of channel delays correspond to two degrees of parallelism in the processing of signals in the channel. The meaning of `delay` is that *no parallel transmission* of signals is made in the infrastructure represented by the SDL channel. For example, an SDL channel representing a data link layer connection (in the OSI stack) – e.g. on an ethernet link – exhibits this type of delay, as the transmission of a signal does not begin until all previous signals have been conveyed. The meaning of `pipeline` is that *fully parallel processing* of signals is made in the infrastructure represented by the SDL channel. For example, a TCP link between two remote hosts connected by a multi-node network path exhibits (asymptotically) `pipeline` delays, as the transmission of a signal may begin as soon as the transmission of the previous signal was initiated, and does not have to wait until the previous signals arrives at destination. (In reality, in this case too a small part of the signal transmission is made sequentially, but that is negligible compared to the end-to-end transmission time).

6.2.4 Example of extended specification

In this section we present a small example illustrating the SDL extensions introduced previously. This is an incomplete version of the SpaceWire protocol specification [sWG00], which is discussed in more depth in Chapter 9.

The Exchange Level of SpaceWire is a data link protocol providing services like connection establishment, error detection and flow control. This level is materialized by a Link Interface that makes the connection between a host system and a physical SpaceWire link. A Link Interface is composed of three entities: a Receiver (RX), a Transmitter (TX) and a State Machine (SM), described in more detail in Chapter 9.

For some of the functions provided by the Link Interface, the SpaceWire standard [sWG00] contains requirements concerning timing. There are two types of requirements:

- requirements concerning various *timeout periods* used by the system. The particularity of SpaceWire is that the normative values for these timeouts are not fixed, but may vary within (large) intervals specified by the standard.

  For example, a *disconnection timer* is used for detecting transmission problems on the physical link. If no signal from the link reaches the receiver for a period equal to the duration of the timer, the *Receiver* informs the *State machine* about the disconnection. The disconnect timeout period may vary between 740 ns and 1080 ns.

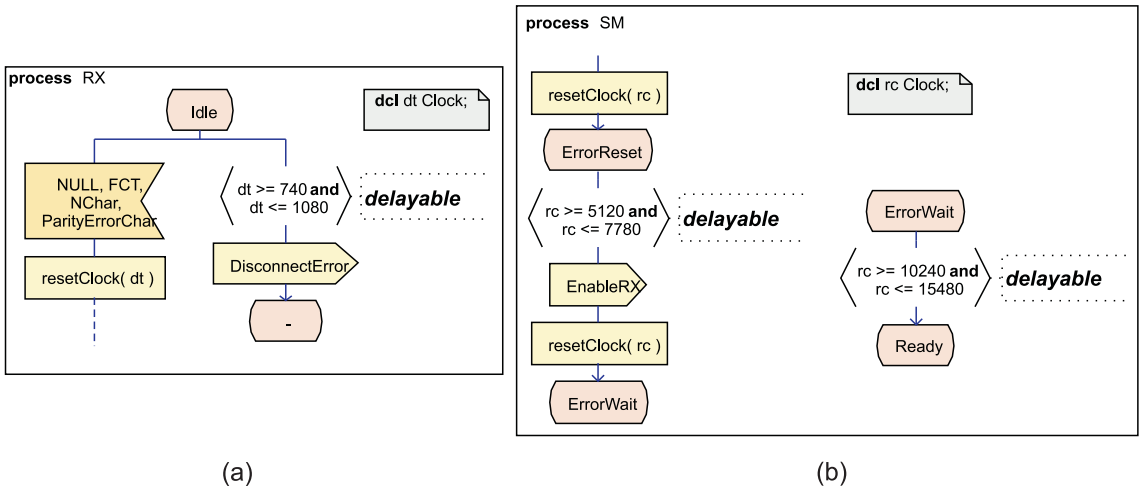


Figure 6.2: Modeling time non-determinism in the SpaceWire specification

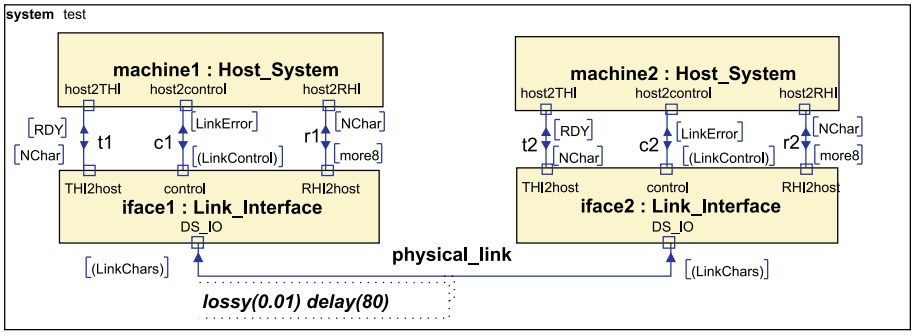


Figure 6.3: The SpaceWire model for validation

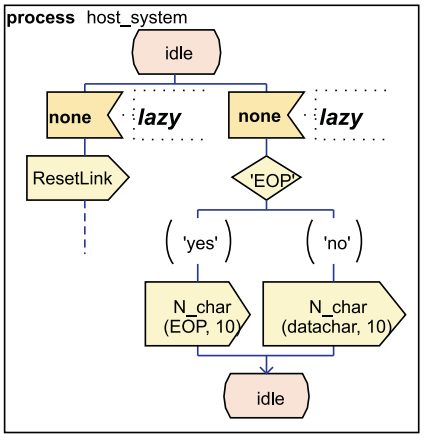


Figure 6.4: Non-deterministic behavior of the host systems

Standard SDL timers are set with a unique duration. Thus, using an SDL timer for modeling the disconnection timeout does not allow to validate the protocol specification for all the possible combinations of timeout values at the two ends of a link. With the extensions introduced in this chapter, the disconnection timeout may be modeled using a clock and a *delayable* transition in the receiver component, as shown in Fig. 6.2-a.

Similarly, the connection establishment phase uses a time-controlled reset cycle, during which the link interface passes successively through several states (the meaning of these states is not important at this stage). The standard specifies the time periods for which the link must remain in each state. For example, the *ErrorReset* state is left after 5.12 μs to 7.78 μs, and the *ErrorWait* state is left after 10.24 μs to 15.48 μs. Such non-deterministic time requirements may also be modeled using clocks and delayable transitions, as shown in Fig. 6.2-b.

– requirements concerning the *speed* (and other characteristics) of the physical link. In an SDL model, the *physical link* is modeled as a channel between the two *link interfaces*. A link operating at 100 Mbps transfers a bit in 10 ns. If the character encoding layer is abstracted away in the SDL model, and full characters are considered to be sent on the link, the delay for a character is 80 ns[1]. This is shown in the model in Fig. 6.3.

  The *lossy* attribute may be used to model the fact that a physical link is unreliable (message distortion is not taken into account in this case).

The model shown in Fig. 6.3 is built for validating the specification of the SpaceWire link interface. For this, the context in which the link operates also needs to be modeled. This context includes the two *host systems* operating the connected interfaces. Their complete behavior should not be specified; nevertheless, several properties of *host systems* have to be taken into account for validation purposes:

– a host system may fail and reset the link at random,

– a host system may send characters on the link at random.

Fig. 6.4 shows how this non-deterministic behavior is modeled using *lazy* transitions.

6.3 Impact of extensions on the ASM semantics of SDL

The purpose of this section is to study how the timing extensions introduced in §6.2 can be integrated in the ASM semantic framework of SDL [IT99c], and to provide a precise understanding of the extensions for readers familiar with the formal SDL semantics. We do not aim to give here a complete list of modifications to be made to Z.100 Annex F [IT99c], but rather to show the main lines for the implementation of the extensions in ASM.

There are several points to be detailed:

– the handling of *explicit clocks* in ASM,

– the handling of action execution durations, communication delays and timers in ASM,

– the introduction of a notion of *controlled time*, which progresses depending on the state of the system.

[1] As a character may have different lengths, this requirement is actually represented differently in the detailed model presented in Chapter 9.

6.3.1 Explicit clocks

Explicit clocks introduce two new predefined data types: `Clock` and `DifClock`. The definition of these types introduce many changes in the ASM semantics, as can be seen from the definition of other predefined types (e.g. `boolean`, `integer`, etc. See [IT99c] Part 3, Ch. 3.). We do not detail all the ASM constructs involved in the definition of these types here, especially as they do not provide insight into the functioning of the timed SDL model. Instead, we concentrate on the next sections which describe the handling of delaying channels, timers and time in the ASM model.

The predefined types `Clock` and `DifClock` introduce two new data domains:

$SDLClock =_{\text{def}} Clock \times Identifier$

$SDLDifClock =_{\text{def}} Real \times Identifier$

which are included in the *Value* domain (see [IT99c] Part 3, §2.1.3.1) gathering values of all predefined data types:

$$\begin{aligned} Value \quad =_{\text{def}} \quad & SDLClock \cup SDLDifClock \cup \\ & SDLInteger \cup SDLBoolean \cup SDLReal \cup SDLCharacter \cup SDLString \cup PId \cup \\ & Object \cup SDLLiterals \cup SDLStructure \cup SDLArray \cup SDLPowerset \end{aligned}$$

Values from the *SDLClock* domain refer to elements from a domain called *Clock*:

controlled domain *Clock*

initially $Clock = \varnothing$

The *Clock* domain gathers all the clocks defined in the SDL system (both implicit and explicit) at a specific moment. Explicit clocks are added whenever a data item of the type `Clock` is created (i.e. at agent creation time for static clocks, or when the operator `mkClock` is called for dynamic clocks). Implicit clocks are added whenever the underlying semantics needs them (e.g. for measuring signal transmission times), as will be shown in the next section.

The function *clockValue* gives the current value of a clock:

controlled *clockValue* : $Clock \rightarrow Real$

Several ASM artifacts are needed to make the `Clock` and `DifClock` types functional. For example, the function *compute* ([IT99c], Part 3, §3.1) must be able to compute the predefined operators of the new types: `"<"`, `"<="`, `"="`, `">="`, `">"`, `"-"` for `Clock`, `"<"`, `"<="`, `"="`, `">="`, `">"` for `DifClock`. We show below the definition of the functions used to compute the predefined operators, which have to be called from *compute*:

computeClock(*procedure*: *Procedure*, *values*: *Value**) : *Value* $=_{\text{def}}$
  if *procedure.procName* = "−" **then**
    mk-*SDLDifClock*(*values*[1].s-*Clock.clockValue* − *values*[2].s-*Clock.clockValue*,
      DifClockType)
  else let *val1* = *values*[1].s-*Clock.clockValue*, *val2* = *values*[2].s-*Nat* **in**
    case *procedure.procName* **in**
    | "<" : **mk-***SDLBool*(val1 < val2, *BooleanType*)
    | "<=" : **mk-***SDLBool*(val1 ≤ val2, *BooleanType*)
    | "=" : **mk-***SDLBool*(val1 = val2, *BooleanType*)
    | ">" : **mk-***SDLBool*(val1 > val2, *BooleanType*)
    | ">=" : **mk-***SDLBool*(val1 ≥ val2, *BooleanType*)
    endcase

endlet

endif

computeDifClock(*procedure*: *Procedure*, *values*: *Value**) : *Value* =$_{\text{def}}$

let *val1* = *values*[1].s-*Real*, *val2* = *values*[2].s-*Nat* **in**

case *procedure.procName* **in**

| "<" : **mk-***SDLBool*(val1 < val2, *BooleanType*)

| "<=" : **mk-***SDLBool*(val1 ≤ val2, *BooleanType*)

| "=" : **mk-***SDLBool*(val1 = val2, *BooleanType*)

| ">" : **mk-***SDLBool*(val1 > val2, *BooleanType*)

| ">=" : **mk-***SDLBool*(val1 ≥ val2, *BooleanType*)

endcase

endlet

endif

The `resetClock` construct introduced in SDL is implemented in the ASM semantics by a new behavior primitive, similar to the behavior primitives defined in the standard for each basic action (output, assignment, set, reset, etc.). A `resetClock` action is represented by an element of the *ResetClock* domain:

ResetClock =$_{\text{def}}$ *ValueLabel* × *ContinueLabel*

where the *ValueLabel* refers a *SDLClock* value. The reset is then realized by the following macro:

EVALRESETCLOCK(a : *ResetClock*) ≡

value(a.s-*ValueLabel*,*Self*).*clockValue* := 0

Self.currentLabel := a.s-*ContinueLabel*

6.3.2 Execution and communication delays. Timers

Execution delays are handled using implicit clocks. However, the handling of these clocks need not be defined by the semantics, as in §6.2.2 we have shown the syntactic transformation of a time consuming action into an implicit state, with a delayable transition and an additional implicit clock. This transformation (see the *Transformation* step in §3.3.1, Fig. 3.4) may be handled in the static semantics, and the dynamic semantics will handle the newly introduced clock like any explicit clock[2].

For handling *communication delays* and *timers* in the ASM semantics, we have two options:

1. to adapt the *schedule* mechanism already existing in the standard (see §3.3.3, page 56), or
2. to use implicit clocks for measuring communication and timer delays.

We will discuss both alternatives in the following paragraphs.

[2]The dynamic semantics must nevertheless ensure mutual exclusion for alternating agents (i.e. sub-agents of a *process*). In the context of time consuming actions, this means that an alternating agent should not yield the *execution rights token* (modeled by the *isActive* function in the standard formal semantics) upon entering an implicit action state.

ASM implementation using schedules

Timers and communication delays are implemented in the standard ASM semantics of SDL using the *schedule* mechanism. This mechanism can be easily adapted to accommodate channels with specified delays, such as those introduced in §6.2.3. We define the following derived functions which retrieve the attributes from a channel specification[3]:

channelDelayKind(*l* : *Link*) : *Channel-delay-kind* =$_{\text{def}}$
    l.nodeAS1.s-*Channel-delay-kind*

channelMinDelay(*l* : *Link*) : *Nat* =$_{\text{def}}$
    l.nodeAS1.s-*Channel-min-delay*

channelMaxDelay(*l* : *Link*) : *Nat* =$_{\text{def}}$
    l.nodeAS1.s-*Channel-max-delay*

channelLossy(*l* : *Link*) : *Boolean* =$_{\text{def}}$
    l.nodeAS1.s-*Channel-loss-probability* $\neq 0$

A controlled function is necessary to hold the delivery time of the previously delivered signal, for **delay** channels. This is because in **delay** channels, unlike in **pipeline**, the delivery of a signal is done only after the previous signal has reached the destination.

controlled *previousDeliveryTime* : *Link* $\rightarrow$ *Time*

The FORWARDSIGNAL macro (already defined in [IT99c] Part 3, §2.1.1.3), which is executed by channel ASM agents (*Link*) and is responsible for delivering signals between the ends of a link, is modified to take into account the channel delay and loss attributes:

```
FORWARDSIGNAL ≡
  if Self.from.queue ≠ empty then
    let si = Self.from.queue.head in
      if Applicable(si.signalType, si.toArg,si.viaArg,Self.from,Self) then
        if (Self.channelDelayKind = "pipeline" ∨
        (Self.channelDelayKind = "delay" ∧ Self.previousDeliveryTime ≤ now)) then
          DELETE(si,Self.from)
          choose looseIt : looseIt ∈ Boolean
            if ¬Self.channelLossy ∨ ¬looseIt then
              INSERT(si,now+Self.delay,Self.to)
              si.viaArg := si.viaArg \
                  Self.from.nodeAS1.nodeAS1ToId, Self.nodeAS1.nodeAS1ToId
            endif
          endchoose
          Self.previousDeliveryTime := now + Self.delay
        endif
      endif
    endlet
  endif
```

[3]The function definitions use the *nodeAS1* function, which makes the interface between the ASM behavior description objects (defined in the dynamic semantics), and the ASM representation of the SDL syntax (defined in the static semantics).

The signal delivery algorithm described above, as well as the one described in [IT99c], makes use of a monitored function (*delay*) which gives the delay applied to a specific signal instance traveling through the link:

monitored *delay* : *Link* → *Duration*

As in [IT99c], the above algorithm preserves the order of transferred signals under specific assumptions about the values of *delay*. The assumptions are given in [IT99c] in the form of integrity constraints on *delay*. Adapted for our semantics, the integrity constraints on *delay* can be formulated as follows: whenever the macro FORWARDSIGNAL is executed, if the updates (i.e. ASM assignments) specified in the macro are executed, the following inequalities must hold:

$$Self.channelMinDelay \leq Self.delay \leq Self.channelMaxDelay$$

$$Self.previousDeliveryTime \leq now + Self.delay$$

The first constraint ensures that the link delay is within the specified bounds. The second constraint ensures that order is preserved on **pipeline** channels (on delay channels, it is preserved by definition).

In order to preserve the strict nature of delay specifications, the execution of the FORWARDSIGNAL macro must be considered *eager* whenever there is a signal that can be effectively transferred by the link. This issue is taken into account in §6.3.3, when the semantics of time progress is described.

ASM implementation using implicit clocks

The behavior of delaying channels and timers may alternatively be described in ASM using implicit clocks, instead of the *schedule* mechanism. A reason for using implicit clocks is that the schedule mechanism uses absolute times: signals (and timers) are stamped with an absolute arrival time, which is then compared with the absolute clock *now* (see the definition of the function *queue* from [IT99c], Part 3, §2.1.1.2) to implement signal arrival. Such a use of absolute time marks in a model poses important problems for model checking, as there are currently no abstractions able to reduce an infinite state space generated by such a model to a finite representation, in the general case.

For SDL systems which use only relative times (i.e. timers set with statements like `set(now+d,t)`, where `d` is a duration not depending on the value of **now**) as well as delaying channels, the use of absolute time marks may be avoided. In this paragraph we discuss a method of handling *relative timers* and *delaying channels* in the ASM semantics of SDL, using only constructs which can be mapped to timed automata primitives (i.e. clocks, guarded transitions, urgency). We conjecture that the resulted semantics is equivalent to the timed automata-based semantics of SDL examined later on in §6.4[4].

In this implementation, *Link* agents hold a signal queue:

controlled *linkQueue* : *Link* → *SignalInst**

(We note that the ASM operators for list creation and concatenation are respectively the brackets $< ... >$ and $^\frown$. This notation is used in the macros defined below.)

A sequence of *Clock*s is used to measure the travel time of signals. In case of **pipeline** links, there is a clock for each signal in the queue. In case of **delay** links, only one clock (corresponding to the first signal in the queue) is sufficient.

[4]The proof of the equivalence between the two semantics may be based on an argument of strong bisimulation between the LTSs generated by the two semantics. However, due to the complexity of the SDL language and of the semantics, the actual realization of the proof is hardly possible.

controlled *linkQueueClocks* : *Link* → *Clock**

The macro ForwardSignal (from [IT99c], Part 3, §2.1.1.3) is redefined as shown below. Every link executes (in parallel, and whenever possible) two actions:

1. *retrieve* messages from the head of the queue of the transmitting end, and place them in the *linkQueue*,
2. *deliver* messages from the head of the *linkQueue* to the receiving end, when their arrival time has come.

```
ForwardSignal ≡
    RetrieveSignal
    DeliverSignal

RetrieveSignal ≡
    if Self.from.queue ≠ empty then
        let si = Self.from.queue.head in
            if Applicable(si.signalType, si.toArg,si.viaArg,Self.from,Self) then
                Delete(si, Self.from)
                Self.linkQueue := Self.linkQueue ^ <si>
                if Self.channelDelayKind = "pipeline" ∨ Self.linkQueueClocks = empty
                then
                    extend Clock with c
                        c.clockValue := 0
                        Self.linkQueueClocks := Self.linkQueueClocks ^ <c>
                    endextend
                endif
            endif
        endlet
    endif

DeliverSignal ≡
    if Self.linkQueue ≠ empty
        ∧ Self.linkQueueClocks.head.clockValue ≥ Self.channelMinDelay
        ∧ Self.linkQueueClocks.head.clockValue ≤ Self.channelMaxDelay
    then
        Insert (Self.linkQueue.head, now, Self.to)
        Self.linkQueue := Self.linkQueue.tail
        if Self.channelDelayKind = "pipeline" then
            Self.linkQueueClocks := Self.linkQueueClocks.tail
        elseif Self.linkQueue.tail ≠ empty then
            extend Clock with c
                c.clockValue := 0
                Self.linkQueueClocks := <c>
            endextend
        else
```

```
        Self.linkQueueClocks := < >
      endif
    endif
```

When a link retrieves a message from the transmitting end, the message is put in the link queue. If the link is **pipeline**, a new clock is created and set to 0, in order to measure the travel time of the newly handled signal; the same is true if the link is **delay** and there is no other signal currently traveling through it. The link delivers a signal at the other end only when it is in the head of the queue and the corresponding clock satisfies the channel delay bounds. The clock corresponding to the delivered signal is erased from the queue, but if the link is **delay** a new clock is created for the next signal in the queue (if there is one).

We note that with this semantics of channels, the mechanism of *schedules* from [IT99c], Part 3, §2.1.1.2, which are used to delay signal arrival is no longer necessary. For simplicity, in the macros described above we use the same basic primitives (INSERT, DELETE, *queue*) as in [IT99c], but each time a signal inserted in a gate schedule (i.e. INSERT is called) the signal is stamped with *now* and not with a delayed arrival time.

Timers can be handled similarly, using clocks instead of the *schedule*. In [IT99c], timers are modeled using two domains:

1. the timer definitions: $Timer =_{\text{def}} \{tid \in Identifier : tid.idToNodeAS1 \in Timer\text{-}definition\}$
2. the timer instances: $TimerInst =_{\text{def}} PId \times Timer \times Value^*$.

The same domains are used in this variant of the semantics. An additional function maps every (non-expired) timer instance to a clock:

controlled *timerClock* : *TimerInst* → *Clock*

Another function keeps the non-expired timer instances of every agent:

controlled *runningTimers* : *SDLAgent* → *TimerInst*-**set**

A third function keeps the relative deadline of each (non-expired) timer instance. We will consider that the relative duration is a *natural*, for every timer set by the system. This is a restriction to the SDL language, made for simplifying the compatibility with timed automata.

controlled *timerDeadline* : *TimerInst* → Nat

The macros SETTIMER and RESETTIMER described in [IT99c] are modified as shown below:

```
SetTimer(tm:Timer, vSeq :Value*, t:Time) ≡
  let tmi = mk-TimerInst(Self.self, tm, vSeq ) in
    Delete(tmi, Self.inport)
    Self.runningTimers := Self.runningTimers ∪ { tmi }
    extend Clock with c
      tmi.timerClock := c
      c.clockValue := 0
    endextend
    if t = undefined then
      tmi.timerDeadline := tm.duration
    else
      tmi.timerDeadline := t − now
    endif
  endlet
```

RESETTIMER(*tm*: *Timer*, *vSeq* : *Value**) ≡

let *tmi* = **mk-***TimerInst*(*Self.self*, *tm*, *vSeq*) **in**

DELETE(*tmi*, *Self.inport*)

Self.runningTimers := *Self.runningTimers* \ { *tmi* }

endlet

The derived predicate *Active* ([IT99c], Part 3, §2.1.1.5), which indicates whether a timer is active or not is redefined as:

Active(*tmi* : *TimerInst*) : *Boolean* =$_{\text{def}}$

$tmi \in Self.runningTimers \vee tmi \in Self.inport.schedule$

As can be seen above, the macro SETTIMER does not handle the delivery of the timer message in the agent input port, as it does in the semantics using *schedules*. Therefore, an additional rule macro is described to that end:

EXPIRETIMERS =

choose *tmi* : $tmi \in Self.runningTimers \wedge$

$tmi.timerClock.clockValue \geq tmi.timerDeadline$

INSERT(*tmi*,*now*,*Self.inport*)

endchoose

The macro EXPIRETIMERS must be inserted in the normal execution cycle (described in §2.3.2.2 of the standard semantics) of every *SDLAgent*. There are several phases of the cycle in which it can be inserted with the same effect, e.g. it may be called within the FIRETRANSITION macro.

6.3.3 Controlled time

An important change brought by the timing extensions in the semantics of SDL is the handling of time progress. As we mentioned before, in order to guarantee the satisfaction of timing constraints, time must be modeled as an internal, *controlled* parameter of the system, instead of being an environment parameter.

Concretely, this means that the function *now*, instead of being a monitored function as in [IT99c], is modeled as a controlled function:

controlled *now* : → *Real*

initially *now* = 0

A new ASM agent is responsible for handling the *now* function and the system clock values (the *clockValue* function). This agent exists from system creation until the end of system execution.

TimeAgent =$_{\text{def}}$ *Agent*

static *timeAgent* : → *TimeAgent*

initially *Agent* = { *system*, *timeAgent* }

The program of the time agent consists in computing the maximal value up to which time may advance at a moment, and then advance *now* and the values of all system clocks by an amount non-deterministically chosen between 0 and the computed maximal value:

```
ADVANCETIME ≡
  choose v : v ∈ Real ∧ 0 ≤ v ∧ v ≤ maxTimeProgress
      now := now + v
      do forall c : c ∈ Clock
        c.clockValue := c.clockValue + v
      enddo
  endchoose
```

The key of the above specification is the *maxTimeProgress* function, which implements the *time progress conditions* (which are similar to those of time automata with urgency):

- Time does not progress while an SDL agent is executing a transition. We remind that in the model executed by the dynamic semantics, time consuming actions are already transformed into waiting states and zero-time actions.
- Time does not progress if there is an agent in a stable state for which an eager transition is fireable.
- Time can progress with at most d time units, if there is an agent in a stable state for which a delayable transition is fireable, and progress beyond d would disable the transition.
- Time can progress with at most d time units if there exists a signal on a delayable channel which should arrive at the end of the channel in at most d time.
- Time can progress with at most d time units if there exists a running timer is a process which should expire in d time.

The definition of *maxTimeProgress* is then:

$$maxTimeProgress : Real\text{-}inf =_{\text{def}} min(< maxTimeZeroActions, maxTimeEager, maxTimeDelayable, maxTimeDelayLinks, maxTimeTimers >)$$

In the definition, we use the domain:

$$\textbf{static } Real\text{-}inf = Real \cup \{ \infty \}$$

where **static** $\infty :\rightarrow X$ is a distinct element of the ASM basic set denoting the infinite value. We also suppose that the function

$$min(\ Real\text{-}inf^* \cup Real\text{-}inf\textbf{-set}) : Real\text{-}inf$$

returns the minimum of a sequence or set of real numbers including ∞, and that a function "$\leq$" : $Real\text{-}inf \times Real\text{-}inf \rightarrow Boolean$ extends the comparison operators for ∞ in the usual way.

In what follows, we examine the functions involved in the definition of *maxTimeProgress*.

Zero execution time for transitions

To ensure that time does not progress while an agent is executing a transition, until it reaches a stable state, the function *maxTimeZeroActions* returns 0 whenever an SDL agent is executing a transition, and ∞ otherwise.

The execution of an SDL agent is a process comporting multiple phases (*execution start*, *transition execution*, *transition selection*, *execution stop*) with complex control and sub-phases. Each phase consists of one or more ASM agent steps (applications of the ASM program). A possible definition of *maxTimeZeroActions* is to allow time to progress only in a particular phase

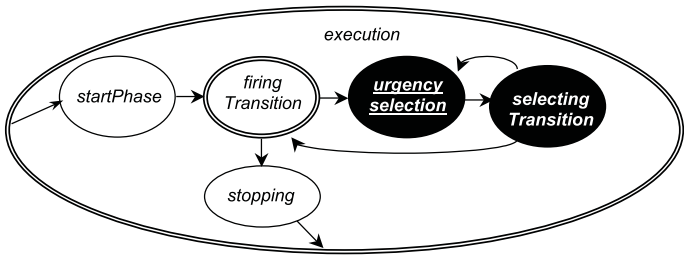


Figure 6.5: Activity phases of SDL agents.

of execution of the ASM agent implementing the SDL agent. In this way, all the other phases in the execution of the ASM agent, which are invisible on the SDL level, are considered atomic with respect to time (i.e. *eager*). The following definition of *maxTimeZeroActions* allows time to progress only when all agents are about to begin the selection of fireable transitions[5]:

maxTimeZeroActions : *Real-inf* $=_{\text{def}}$
if $\forall\, ag \in Agent$: ($(ag.program$ = AGENT-PROGRAM) $\Rightarrow$
$(ag.agentMode1 = execution \wedge$
$ag.agentMode2 = selectingTransition \wedge$
$ag.agentMode3 = startSelection$))
then ∞ **else** 0 **endif**

There is a problem with this definition: in each ASM agent, the cycle between *transition selection – transition execution* is a continual process, i.e. if a selection phase does not find any fireable transition, another selection phase begins unconditionally. Moreover, the cycles of the agents composing the system do not synchronize with each-other. Therefore, with the definition of *maxTimeZeroActions* given before, the system may enter a timelock in which all agents cycle on the transition selection phase, but do not pass through the state tested by *maxTimeZeroActions* simultaneously.

A solution to this problem is to synchronize the execution cycles of all *SDLAgent*s in the system (e.g. using a semaphore variable), so that they are forced to pass through the state tested by *maxTimeZeroActions* simultaneously. This may be done in such a way that the generality of the model is not affected, i.e. all possible interleaving of SDL actions in concurrent agents are preserved. For that, it is necessary to allow the re-execution of the selection phase in an agent even if a fireable transition was found, so that the synchronization introduced above does not *force* the firing of transitions. In the next section we show how we redefine the execution cycle of an *SDLAgent* to take into account these facts.

Eager and delayable transitions

Several changes are needed in order to introduce the conditions on time progress imposed by the existence of eager and delayable transitions into the ASM model. Fig. 6.5 shows the execution cycle of an *SDLAgent* from the standard formal semantics [IT99c], with an additional *urgency selection* phase. In this phase, the clock constraints of the form $x \sim \mathbf{c}$ ($\sim \in \{<, \leq, =, \geq, >\}$), guarding the *eager* and *delayable* transitions which are otherwise fireable (i.e. except for the truth value of these clock guards), are gathered together in two lists:

[5]The test *ag.program* = AGENT-PROGRAM is used in *maxTimeZeroActions* to select the *SDLAgent*s from the whole set of ASM *Agent*s which also contains *SDLAgentSet*s and *Link* agents

controlled *eagerGuards* : → (*Value* × *Nat* × *Procedure*)*

controlled *delayableGuards* : → (*Value* × *Nat* × *Procedure*)*

The components of the tuples kept in these lists represent respectively the referenced clock (x), the constant (**c**) and the comparison operator ($\sim$).

If the clock guard corresponding to an *eager* transition evaluates to *True* with the current values of clocks, then an additional location denoted by the function *trueEagerGuard* is put to *True*:

controlled *trueEagerGuard* : → *Boolean*

The ASM implementation of the *urgency selection* phase is similar to that of the *selecting transition* phase. The differences are that in *urgency selection*, the evaluation of the *enabling conditions* differentiates conditions on clocks from other boolean terms, and constructs the lists mentioned above instead finding fireable transitions (without returning on the first transition found but by traversing all possibly fireable transitions).

We note that in order for the *urgency selection* phase to work properly, all *SDLAgent*s in the ASM model must start executing it in a synchronized manner. Otherwise, if some *SDLAgent*s are still in the *firing transition* phase while others begin the *urgency selection*, the former may change the fireable transitions of the latter "on the fly", thus invalidating the global time progress condition. Synchronization may be achieved through an ASM variable used as semaphore, like in the case of the *selecting transition* phase (see previous section). Thus, the phases represented in black in Fig. 6.5 are phases at the beginning of which *SDLAgent*s need to synchronize.

With these preparations, the expressions of the functions *maxTimeEager* and *maxTimeDelayable* are:

maxTimeEager : *Real-inf* =$_{\text{def}}$
  if *trueEagerGuard* **then**
    0
  else
    min(< *v*.s-*Nat* − *v*.s-*Value*.s-*Clock.clockValue* | *v* **in** *eagerGuards* :
      v.s-*Nat* − *v*.s-*Value*.s-*Clock.clockValue* ≥ 0 ∧
      (*v*.s-*Procedure.procName* = ">" ∨
      v.s-*Procedure.procName* = ">=" ∨
      v.s-*Procedure.procName* = "=")
    >)
  endif

maxTimeDelayable : *Real-inf* =$_{\text{def}}$
  min(< *v*.s-*Nat* − *v*.s-*Value*.s-*Clock.clockValue* | *v* **in** *delayableGuards* :
    v.s-*Nat* − *v*.s-*Value*.s-*Clock.clockValue* ≥ 0 ∧
    (*v*.s-*Procedure.procName* = "<" ∨
    v.s-*Procedure.procName* = "<=" ∨
    v.s-*Procedure.procName* = "=")
  >)

In the above definitions, we suppose that *min* returns ∞ if the list or the set parameter is empty.

As can be seen, *maxTimeEager* returns 0 if the "urgency selection" phase has found (at least) one *eager* transition which is fireable with the current values of clocks and **now**. If no

such transitions are found, the list *eagerGuards* contains the (atomic clock comparisons from the) guards of *eager* transitions which may be *enabled* by only letting time pass. For every such transition, time may not progress beyond the *lower* bound of the transition guard, as this would mean to let time progress while the eager transition would be enabled. Actually, *maxTimeEager* cannot handle eager transitions with guards of the form $x > \mathbf{c}$. However, such transitions are logically nonsense, as they allow time to progress up to $x = \mathbf{c}$ – where they are not enabled, but not beyond – where they would be enabled. Therefore, such transitions should be signaled as modeling errors.

maxTimeDelayable is based on the list *delayableGuards*, computed in the "urgency selection" phase, which contains the (atomic clock comparisons from the) guards of *delayable* transitions which may be *disabled* by letting time pass. For every such transition, time may not progress beyond the *upper* bound of the transition guard, as this would mean to let time progress until the transition becomes disabled.

Channel delays and timer expiration

The implementation of time progress conditions related to delaying channels and timers in ASM depends on whether these constructs are implemented using the *schedule* mechanism or using *implicit clocks* (see §6.3.2).

1. *Time progress conditions for schedules.* A schedule does not let time pass beyond the arrival time of the first signal *not arrived*. Globally, the maximum amount of time that is allowed (by the schedules) to pass is the minimum of the amounts let by all schedules in the system. Moreover, in the implementation using schedules, there is no difference between the time progress condition generated by timers and that generated by delaying channels. We have therefore the following definitions for *maxTimeDelayLinks* and *maxTimeTimers*:

   $$\begin{aligned}
   &maxTimeDelayLinks : \text{Real-inf} =_{\text{def}} maxTimeSchedules \\
   &maxTimeTimers : \text{Real-inf} =_{\text{def}} maxTimeSchedules \\
   &maxTimeSchedules : \text{Real-inf} =_{\text{def}} \\
   &\quad min(\{\ g.signalsInTransit.head.arrival - now \mid \\
   &\qquad g \in Gate : g.signalsInTransit \neq empty\ \})
   \end{aligned}$$

   where the function *signalsInTransit* computes the list of signals in a gate schedule which are not yet arrived:

   $$signalsInTransit(g : Gate) : SignalInst^* =_{\text{def}} < si \textbf{ in } g.schedule : (now < si.arrival) >$$

2. *Time progress conditions for implicit clocks.* If implicit clocks are used for handling delaying channels and timers in the ASM semantics, the time progress conditions are based on the values of implicit clocks and the relative timer and signal delays.

   The time progress condition imposed by delaying channels (*maxTimeDelayLinks*) are:

   - If there is a signal at the transmitting end of a *Link*, which is ready to be put in the link signal queue (action RetrieveSignal, defined on page 109), then time may not advance until this action is done.
   - Otherwise, time may advance until the (maximum) arrival time of the first signal which is to reach its destination.

   We get the following definition for *maxTimeDelayLinks*[6]:

[6] The test *lk.program* = Link-Program is used to select the ASM agents implementing *Link*s.

$maxTimeDelayLinks$: *Real-inf* $=_{\text{def}}$
  $min(\ \{\ lk.channelMaxDelay - lk.linkQueueClocks.head.clockValue\ |$
    $lk \in Agent : lk.program = \text{LINK-PROGRAM} \wedge lk.linkQueue \neq empty\ \}\ )$

The time progress condition imposed by the handling of timers using implicit clocks is that time cannot advance beyond expiration moment of the timer with the closest deadline[7]:

$maxTimeTimers$: *Real-inf* $=_{\text{def}}$
  if $\exists\ ag \in Agent : ag.program = \text{LINK-PROGRAM} \wedge ag.from.queue \neq empty$
  then
    0
  else
    $min(\{min(\{tm.timerDeadline - tm.timerClock.clockValue \mid tm \in ag.runningTimers\ \}\ )\ |$
      $ag \in Agent : ag.program = \text{AGENT-PROGRAM}\})$
  endif

Conclusion

For an increased precision in the definition of the extensions proposed for SDL, in this section we have described the semantics of the extensions in ASM, and we have adapted the ASM semantics of time to better suit the needs of timing analysis (especially by modeling time as a *controlled* parameter of the system). ASM provides a formal operational description of the semantics, which it can capture at the right abstraction level.

However, while an ASM model is semantically equivalent with a labeled transition system, the direct application of model checking techniques on multi-agent ASM specifications is complicated by the fine granularity of ASM transitions, and by the high level of asynchrony between ASM agents. Moreover, current SDL tools such as [TEL00a, TEL00b] do not use the standard semantics but rather some simplified LTS-based semantics, which avoids the problems of ASM mentioned above and thus reduce the state space explosion problem. For these reasons, we find it easier to adapt timed automata analysis techniques to SDL, which is one of the goals of this work, by taking as starting point the simplified semantics implemented by SDL tools.

In the following section, we discuss the impact of our proposed extensions on the semantics of SDL as given by tools. We obtain thus a semantic model for SDL which is closer to the timed automata model.

6.4 Impact of extensions on the LTS-based semantics of SDL

Simulation and verification tools for SDL, like [TEL00a, TEL00b], work by building an LTS corresponding to the possible executions of an SDL model. This LTS complies only partially to the standard ASM semantics of SDL, as some simplifications are made for efficiency reasons, such as: transitions are considered atomic (interleaving is at the level of transitions and not of individual actions), transitions take 0 time units, time progress is controlled in a simplified manner.

In this section we examine how the extensions introduced in §6.2 can be integrated in the semantics of SDL implemented by the *Object*GEODE simulation tool [TEL00a]. Many details

[7] The test $ag.program = \text{AGENT-PROGRAM}$ is used to select the ASM agents implementing SDL agents.

are however treated only informally, as a more rigorous definition was given in ASM in the previous section. As we noted previously, we may only conjecture that the semantics described in the following is equivalent with the ASM semantics using implicit clocks, described in the previous section. A proof for this equivalence is practically impossible due to the complexity of SDL and of the semantics.

The *Object*GEODE discrete time semantics of SDL

For a given SDL system $\mathcal{S}$, the *Object*GEODE verification tool builds a labeled transition system $\mathcal{G}_\mathcal{S}$ corresponding to the state space of $\mathcal{S}$, which is used to check behavioral properties. The nodes of $\mathcal{G}_\mathcal{S}$ are global states of the SDL system, comprising the discrete state of each process instance, procedure or service, the values of all variables, the content of all queues, as well as the *relative delay* until expiration for each active timer in the system. The predefined variable **now** is not part of the system state[8], since this would cause $\mathcal{G}_\mathcal{S}$ to be infinite systematically. The initial state of the LTS, q_0, corresponds to the state of the SDL system after the initial creation of all statically declared SDL agents.

$\mathcal{G}_\mathcal{S}$ may contain the following kinds of transitions between states:

1. **internal discrete transition.** $q_1 \xrightarrow{t} q_2$ iff there is an SDL transition identified by t, enabled in the state q_1 and which takes the system into the state q_2. A discrete transition is caused by an *input*, a *priority input*, a *continuous signal*, a *signal save* or *discard*, etc. The modifications of the components of the state are those prescribed by the standard SDL semantics.

2. **feed discrete transition.** $q_1 \xrightarrow{t} q_2$ iff t is an input transition, and the signal that causes it may be sent by the environment of the SDL system.

   It is considered that signals coming from the environment may arrive at whatever moment, and therefore a feed transition $q_1 \xrightarrow{t} q_2$ is enabled even if the signal causing the transition is not actually in the queue, and there are other signals in the queue. For the signal parameters, the modeler has to specify particular values considered to be representative, so that the graph $\mathcal{G}_\mathcal{S}$ is not constructed for all possible combinations of parameter values.

   This implementation of the communication between the SDL specification and the environment is compliant with the standard ASM semantics, in which gate *schedules* are modeled as *shared* ASM functions and can be freely modified by the environment.

3. **time transition.** $q_1 \xrightarrow{time(c)} q_2$ if the next timer to expire has c time units until expiring. q_2 is equal to q_1 except for the delays of active timers, which are decreased by c. A semantic parameter of the tool specifies whether time transitions are allowed at all times, or only when the system is idle (i.e. there is no internal discrete transition and no timeout transition enabled in q_1).

   The above conditions specify how time progress is controlled in the construction of $\mathcal{G}_\mathcal{S}$. Time transitions are interleaved with other SDL transitions, therefore SDL transitions always take 0 time. Moreover, time advances in discrete steps, and always up to the next timer expiration deadline.

[8] In consequence, certain expressions involving **now** cannot be interpreted correctly in $\mathcal{G}_\mathcal{S}$.

4. **timeout transition.** $q_1 \xrightarrow{timeout\ t} q_2$ iff the timer t is active in q_1 and its relative delay until expiration is 0. q_2 is equal to q_1 except for the active status of t and the queue to which the signal t is appended.

Impact of language extensions and continuous time progress conditions

For an SDL specification $\mathcal{S}$ using the timing extensions introduced previously in this chapter, the semantics is given by an LTS $\mathcal{G}_{\mathcal{S}}^{\tau}$ that we call *timed semantic graph*. The states of this LTS have the form $q^{\tau} = (q, q^c, \mathcal{X}, \mathbf{v})$, where:

- q is the global system state, like in $\mathcal{G}_{\mathcal{S}}$. q comprises the discrete state of all processes, procedures and services that are active in the system, as well as the values of all variables and the contents of all signal queues.
- q^c contains the state (i.e. contents) of the queues associated to delaying channels (one queue for each direction of a delaying channel).
- $\mathcal{X}$ is a set of clock identifiers. It contains an identifier for each explicit or implicit clock existing in the current system state. The set $\mathcal{X}$ is variable due to dynamic clock and process creation/deletion, as well as to the handling of implicit clocks.

  Implicit clocks are used for measuring signal delays and for handling timers. For each running timer u, the initial *relative deadline* d_u is held in the state q, like in $\mathcal{G}_{\mathcal{S}}$, but this value is *not* modified by *time* transitions.
- $\mathbf{v}$ is a valuation of the clocks $\mathbf{v} : \mathcal{X} \to \mathbb{R}$. It is similar to the way clock values are handled in the semantic graph of timed automata (§5.3):

The *initial state* of the LTS $\mathcal{G}_{\mathcal{S}}^{\tau}$ is $q_0^{\tau} = (q_0, q_0^c, \mathcal{X}_0, \mathbf{v}_0)$ with q_0 being the initial state from $\mathcal{G}_{\mathcal{S}}$, q_0^c containing void queues for every delaying channel, $\mathcal{X}_0$ containing a clock identifier for each statically declared `Clock` variable in each agent that is created at system startup, and $\mathbf{v}_0 = 0$.

Let $q_1^{\tau} = (q_1, q_1^c, \mathcal{X}_1, \mathbf{v}_1)$ and $q_2^{\tau} = (q_2, q_2^c, \mathcal{X}_2, \mathbf{v}_2)$ be two states of the LTS. The types of edges that can exist in $\mathcal{G}_{\mathcal{S}}^{\tau}$ between q_1^{τ} and q_2^{τ} are described below:

1. **internal or feed discrete transitions** (this category includes both internal discrete transitions and feed transitions). $q_1^{\tau} \xrightarrow{t} q_2^{\tau}$ if:
   (a) t is a discrete transition enabled in q_1, and $q_1 \xrightarrow{t} q_2$ according to the rules in the untimed semantic graph $\mathcal{G}_{\mathcal{S}}$, and
   (b) $\mathbf{v}_1$ satisfies the part of the guard of t referring to clocks (see the language extensions section), and
   (c) q_2^c is obtained from q_1^c by updating the contents of queues of delaying channels with the signals that are *output* during t, and
   (d) $\mathbf{v}_2$ is obtained from $\mathbf{v}_1$ by applying the clock operations (resets, creations) specified on the transition t, and
   (e) $\mathcal{X}_2 = \mathcal{X}_1 \cup \mathcal{X}_{t\uparrow} \setminus \mathcal{X}_{t\downarrow}$ where $\mathcal{X}_{t\uparrow}$ are the clocks created during t and $\mathcal{X}_{t\downarrow}$ are the clocks destroyed during t, and $\forall x \in \mathcal{X}_{t\uparrow}\,,\ \mathbf{v}_2(x) = 0$.
   More precisely, the rules for computing the sets $\mathcal{X}_{t\uparrow}$ and $\mathcal{X}_{t\downarrow}$ are as follows: For each timer u that is *set* by the transition t, a clock x_u is added ($x_u \in \mathcal{X}_{t\uparrow}$). For each

timer u that is *reset* by the transition t, the corresponding clock x_u is deleted from ($x_u \in \mathcal{X}_{t\downarrow}$).

For each output of a signal s, s is placed in the queue corresponding to the first channel on the signal delivery path (which is computed statically, according to the standard semantics of SDL). If that channel is annotated as **pipeline**, a new clock x_s is added ($x_s \in \mathcal{X}_{t\uparrow}$), to measure the signal travel time. If the channel is annotated as **delay**, a clock x_s is added only if there is no other signal in the channel queue.

Moreover, for each new `Clock` object x created using `mkClock()` or as a consequence of the creation of a new agent, $x \in \mathcal{X}_{t\uparrow}$. For each destroyed clock object x, $x \in \mathcal{X}_{t\downarrow}$.

2. **timeout transition.** $q_1^\tau \overset{timeout\ u}{\longrightarrow} q_2^\tau$ if u is active in q_1^τ and the the value of the clock x_u corresponding to the timer u is equal to the initial relative timer deadline d_u kept in q_1 ($\mathbf{v}_1(x_u) = d_u$). q_2 is the same as q_1 except for the status of u (inactive in q_2) and for the agent queue to which the signal u is appended. $q_2^c = q_1^c$, and $\mathcal{X}_2 = \mathcal{X}_1 \setminus \{x_u\}$.

   Timeout transitions are treated as *eager* transitions (see the definition of *time transitions* below).

3. **signal delivery transition.** $q_1^\tau \overset{deliver\ s}{\longrightarrow} q_2^\tau$ if the signal s is in the head of a *channel queue*, and the clock x_s corresponding to the signal satisfies the constraints of the channel ($c_{min} \leq \mathbf{v}_1(x_s) \leq c_{max}$, where c_{min} and c_{max} are the minimum/maximum delays for the concerned channel). The clock x_s is deleted in $\mathcal{X}_2$, and the signal s which is deleted from the corresponding channel queue in q_2^c.

   If the channel is the last one in the signal delivery path, the signal s is simply put in the destination agent queue in q_2. If the channel is not the last one, the signal is forwarded in the queue of the next channel (i.e. q_2^c is updated), by the same algorithm as in the initial signal output (see *discrete transitions*).

   Signal delivery transitions are treated as *delayable* transitions, as it can be seen in the definition of *time* transitions below.

4. **time transition.** $q_1^\tau \overset{time(\delta)}{\longrightarrow} q_2^\tau$ iff the time progress conditions specified below hold. In that case, $q_2 = q_1$, $q_2^c = q_1^c$, and $\mathcal{X}_2 = \mathcal{X}_1$. $\mathbf{v}_2 = \mathbf{v}_1 + \delta$.

   The *time progress conditions* are: $\forall \delta', \delta''$ such that $0 \leq \delta' < \delta'' \leq \delta$

   (a) there is no discrete *eager* transition and no *timeout* transition enabled in $(q_1, q_1^c, \mathcal{X}_1, \mathbf{v}_1 + \delta')$, and
   (b) there is no *delayable* transition and no *signal delivery* transition enabled in $(q_1, q_1^c, \mathcal{X}_1, \mathbf{v}_1 + \delta')$ and disabled in $(q_1, q_1^c, \mathcal{X}_1, \mathbf{v}_1 + \delta'')$.

Relation to timed automata

The timed semantic graph $\mathcal{G}_\mathcal{S}^\tau$ has the same characteristics as the semantic graph of a timed automaton:

- its states are composed of a discrete part – $(q, q^c, \mathcal{X})$ for a state $q^\tau = (q, q^c, \mathcal{X}, \mathbf{v})$, and a part referring to clocks: $\mathbf{v}$.
- its transitions are classified into transitions which act on the discrete part and do not increase the component $\mathbf{v}$ (*internal*, *feed*, *timeout* and *signal delivery* transitions are included

in this category), and *time* transitions which leave the discrete part intact but increase uniformly the component $\mathbf{v}$.

In [Obe99], we have actually shown that for each SDL specification $\mathcal{S}$, a timed automaton $\mathcal{A}_\mathcal{S}$ can be constructed such that the semantic graph of $\mathcal{A}_\mathcal{S}$ is strongly equivalent to the timed semantic graph $\mathcal{G}^\tau_\mathcal{S}$ of $\mathcal{S}$.

However, the actual construction of $\mathcal{A}_\mathcal{S}$ is not important, as the analysis methods designed for timed automata work with (abstractions of) the semantic graph of an automaton, which for $\mathcal{A}_\mathcal{S}$ is equal to $\mathcal{G}^\tau_\mathcal{S}$. Therefore, in the verification tool described in Chapter 8, we apply timed automata abstractions (such as the *simulation graph*, see §5.4) and model checking techniques directly on the $\mathcal{G}^\tau_\mathcal{S}$ graph described above.

6.5 Discussion

We have presented a set of extensions to the SDL language which allow the modeler of real-time systems to specify more precisely the timing of actions and events described in an SDL model. The extensions improve the *expressivity* of the language, in the sense that both events occurring at precise time instants as well as time non-deterministic events may be modeled; both bounded and unbounded temporal non-determinism may be captured in SDL using the extensions. This flexibility in capturing timing constraints is due to the concept of transition *urgency* taken from timed automata [BST98].

The possibility to express time non-determinism in a flexible way is important especially for the validation of abstract or incomplete specifications. In such specifications, it is often the case that the behavior of a component is not fully specified, yet some information is available about the timing of its actions. An example of such specification is provided in §6.2.4.

Other extensions which have been demanded by SDL users (actions with duration, channels with delays) and for which previous proposals existed [Rou98, Die97], are studied in this chapter and given a precise syntactic and semantic definition. Issues like time progress conditions and atomicity in the context of parallel agents, delaying channels and time consuming actions are examined in our timed semantics.

We argue that the *analyzability* of SDL specifications is also improved by using the extensions and the timed semantics introduced in this chapter, for the following reasons:

- The semantics is more accurate with respect to time. Certain scenarios with unrealistic time lines, which are allowed in the standard semantics of SDL are eliminated in the timed semantics, thus reducing the size of the state space.
- A relation between our timed semantics and timed automata exists, which makes it possible to use timed automata analysis techniques in the context of SDL.

Related proposals

The issue of introducing timing annotations into SDL specification has been studied previously, mostly in the context of model-based performance analysis. We mention here two previous approaches:

1. *The* ObjectGEODE *Performance Evaluation Extensions.* The *Object*GEODE simulator implements a series of extensions [Rou98] for modeling and evaluating performance values of SDL specifications. The modeler has the possibility to attach *durations* to actions,

specified by the lower and upper bounds and a probability distribution. Deployment information may also be introduced in the SDL model, by specifying which processes run on the same processor. This information is necessary to determine which other processes are blocked by the execution of an action with duration in a process. The timing information may be used subsequently to make performance measures by intensive random *simulation*.

The extensions from [Rou98] do not give the user full control over time progress. The semantics of time remains the same as the one implemented by the standard *Object*GEODE simulator, modulo time consuming actions. Moreover, the extensions are not intended for verification of temporal properties, and their semantics is not formally specified.

2. *Queuing SDL* (QSDL, [DHHMC95a, Die97]) is another extension of SDL for modeling timing properties of systems with the goal of doing performance evaluation. QSDL introduces an extension for modeling time consuming actions, the REQUEST statement. The difference between QSDL and our extensions is that the execution time of a REQUEST is not specified statically. Instead, REQUESTs are directed towards *queuing machines*, which compute dynamically the processing time of each REQUEST. Queuing machines represent computing resources shared between several agents of an SDL system, and have a series of attributes (like speed, number of processors, scheduling policy) through which their behavior may be adapted to the necessities of the model. Thus, QSDL allows the modeling of congestion due to system overloads.

   QSDL also introduces extensions for modeling communication delays, although the design choices are debatable: delays (fixed values) must be specified for each message **output**, message overtaking is allowed.

   The critiques made to the *Object*GEODE performance extensions remain valid in the case of QSDL: the extensions are not intended for verification and lack a proper underlying semantic framework. The timed semantics of SDL discussed in this chapter could be adapted to QSDL.

Closer to our proposal is the work related to IF [BFG+99], which does not propose extensions to SDL but gives a timed semantics of SDL in terms of an intermediate formalism (IF), whose semantics is in turn based on communicating extended timed automata [Boz99]. Many aspects of the timed semantics of SDL described in this chapter are found also in the SDL to IF mapping from [Boz99].

Our current efforts [BGM+01] go towards defining a set of high-level extensions for specifying timing information, through which the user should be able to capture the types of information occurring frequently in real-time specifications (e.g. cyclic events with period and jitter, or synchronization between events) without having to express them in terms of low level constructs such as clocks and urgencies. These extensions are based on the semantic framework established in [Boz99] and in the present work.

Chapter 7

Timed property description and verification using MSC and GOAL

An important part of the validation methodology for timed systems proposed in this work is concerned with the verification of functional properties including timing aspects, over SDL specifications. In this chapter we examine how such properties may be specified and verified using MSC and GOAL. There are three aspects that need to be studied:

- the *language constructs* for specifying quantitative temporal properties. The existing constructs of MSC-2000 are sufficient for expressing basic timing properties. In the case of GOAL, we propose a small set of extensions which enable the modeling of quantitative timing.
- the *semantics* of the languages. Defining semantics for a property language means stating precisely the conditions in which a property is satisfied (i.e. the *satisfaction relation* between the set of models and the set of properties). In the case of MSC, a first problem is that MSC-2000 lacks a formal semantics for the timing constructs newly introduced in the language. A second problem is that MSC is not a property language, but rather a language for expressing execution traces to which no particular meaning is attached. Later in this chapter we show that several equally justified definitions may be given to MSC satisfaction. In the case of GOAL, the notion of satisfaction is already defined. Therefore, we only need to provide a proper timed semantics for the language extensions that we introduce.
- the *verification* method. For both languages, we provide an algorithmic verification method by reducing the satisfaction problem to a timed automata model checking problem.

In order to give a sound semantic basis to the definition of GOAL and MSC as temporal property languages, we begin this chapter by defining an abstract property model, Timed Property Automata (TPA). TPA is defined at the level of abstraction of timed automata, and uses the main ideas of Timed Büchi Automata (TBA) introduced in [Alu91]. The main difference with TBA is that TPA is *event* oriented rather than *state* oriented. This makes it more suitable as semantic foundation for MSC and GOAL, which are event-oriented languages.

We use the TPA model also for studying the verification problem. The verification method that we describe for TPA may be projected directly at the level of SDL/MSC/GOAL to obtain verification techniques for these languages.

The chapter is structured as follows: §7.1 introduces the TPA formalism. We discuss first some general notions about formal property specification languages, and we continue with the

definition of TPA and with a study of the TPA verification problem. In §7.2 we discuss the possible definitions of MSC satisfaction, and we sketch a TPA-based semantics for a subset of MSC. In §7.3 we introduce a set of extensions which enable the description of timing information in GOAL, and we discuss informally the semantics and verification issues related to GOAL.

7.1 Timed Property Automata

7.1.1 Property specification languages

In this section we introduce some general notions about the property specification languages used in model checking, in order to set the stage for the formalism defined in the next section. This section does not aim to give a thorough introduction to the subject; for that, the reader is referred to a monograph such as [CGP99] or [Hol91].

Linear vs. branching time

Defined in a general way, a property is an assertion about the behavior of a model. As the behavior of an LTS (§5.2) may be regarded either as a set of possible runs or as an execution tree (with the possibly infinite branches representing runs), it is common to distinguish between properties that make assertions about the individual runs (*linear properties*) and properties that make assertions about the tree of runs (*branching properties*).

Thus, for a *linear* property P and a run ρ, it is possible to say whether the run satisfies the property (denoted $\rho \in P$). The property P may be identified with the set of runs that satisfies it. Based only on the set of possible runs of an LTS (and not on its structure), it can be decided whether the LTS satisfies the property or not.

Branching properties, on the other hand, make assertions about the structure of the tree of runs, and therefore two automata with the same set of accepted runs do not necessarily satisfy the same branching properties.

Logic vs. automata-based property specification

Formalisms for representing both branching and linear properties have been proposed in the literature. There are branching and linear variants of temporal logics [Pnu77, CES86, BAMP83], as well as operational methods for specifying *linear* properties using Büchi automata [Bö60]. In this thesis we are concerned with the latter type of specifications, as they are semantically closer to the property languages considered here (MSC and GOAL).

Safety vs. liveness

In linear time, it is common to consider two types of properties [Lam77]:

- *Safety* properties, which assert that a certain (bad) condition does not occur during the execution of a system. If P is a safety property, and ρ is an execution that does not satisfy P, it means that there is a point in ρ where the "bad" condition occurs. Let α be the prefix of ρ up to that point. A characterization of safety properties is that if $\alpha \notin P$, then $\forall \beta$ a continuation of the run α, the concatenation $\alpha.\beta$ does not satisfy P ($\alpha.\beta \notin P$).

  Invariance properties, for example, are safety properties.

– *Liveness* properties, which assert that a certain (good) condition occurs infinitely often during the execution of a system. If P is a liveness property, then every finite execution α of the LTS has an infinite suffix β (not necessarily realizable in the LTS) such that the concatenation $\alpha.\beta$ satisfies P ($\alpha.\beta \in P$).

A topological characterization of linear properties [AS87] shows that any property is the *conjunction* of a safety property and a liveness property.

Safety and liveness properties are specified and verified using different methods. As we show in the next section, in the formalism considered in this chapter *safety* properties are specified by assuming finite automata acceptance conditions for the automaton representing the specification, while *liveness* properties are specified by assuming Büchi acceptance conditions for the specification automaton.

7.1.2 TPA definition

Definition 7.1 (timed property automaton) *A* timed property automaton *(TPA) is a couple* $B = (A, F)$ *where* $A = (\Sigma, \mathcal{X}, Q, q_0, E)$ *is a timed automaton in which* all transitions have *lazy* urgency*, and* $F \subseteq Q$ *is a set of* accepting states*.*

Property satisfaction is defined through two satisfaction relations: *safety* satisfaction and *liveness* satisfaction. The semantics of the accepting states is defined by the satisfaction relations; it is similar to the Büchi acceptance condition in the case of the liveness satisfaction, and to the finite automata acceptance condition in the case of the safety satisfaction.

Correspondence of runs

The *satisfaction relations* between a TA $A = (\Sigma, \mathcal{X}, Q, q_0, E)$ and a TPA $B = (A', F)$ with $A' = (\Sigma', \mathcal{X}', Q', q_0', E')$ are defined based on a correspondence function between the transitions of A and the transitions of A'. Let $\sigma : Q' \times E \to E' \cup \{\epsilon\}$ be a function, such that $\forall q' \in Q'$, $e \in E$, if $\sigma(q', e) \in E'$ then $\sigma(q', e)$ has q' as source state. The symbol ϵ signifies that there is no transition in E' corresponding to a state q' and a transition e; it avoids the definition of σ as a partial function on $Q' \times E$.

Based on the above definition, we define a correspondence function (denoted σ) between runs of A and runs of A'. Let $\rho = (q_0, \mathbf{v}_0) \xrightarrow{\delta_0} (q_0, \mathbf{v}_0 + \delta_0) \xrightarrow{e_0} (q_1, \mathbf{v}_1) \xrightarrow{\delta_1} \ldots$ be a run of A in the canonical form. The corresponding run of A', $\sigma(\rho) = (q_0', \mathbf{v}_0') \xrightarrow{\delta_0'} (q_0', \mathbf{v}_0' + \delta_0') \xrightarrow{e_0'} (q_1', \mathbf{v}_1') \xrightarrow{\delta_1'} \ldots$ begins in the initial state of A', $(q_0', \mathbf{v}_0')$, and is determined by the following transitions:

– Time steps from $\sigma(\rho)$ are identical with those from ρ: $\forall i.\, \delta_i' = \delta_i$. Given a state $(q_i', \mathbf{v}_i')$, the time step $\xrightarrow{\delta_i}$ uniquely determines the next state of $\sigma(\rho)$: $(q_i', \mathbf{v}_i' + \delta_i)$. The validity of the $\xrightarrow{\delta_i}$ time step is ensured by the fact that all transitions of A' are *lazy* (see the definition of TPA) and therefore any amount of time may pass in any state.

– Discrete steps from $\sigma(\rho)$ are determined uniquely by the discrete steps from ρ as follows:

$$e_i' = \begin{cases} \sigma(q_i', e_i) & \text{, if } \sigma(q_i', e_i) \in E' \text{ and } \mathbf{v}_i' + \delta_i \in guard(e_i'), \\ \epsilon & \text{, otherwise.} \end{cases}$$

In the former case, $(q_i', \mathbf{v}_i' + \delta_i) \xrightarrow{e_i'} (q_{i+1}', \mathbf{v}_{i+1}')$ is a usual discrete transition of the timed automaton A'. Its validity is ensured by the fact that $e_i' = \sigma(q_i', e_i) \in E'$ is a discrete

transition departing from q'_i (see definition of σ above) and the guard is satisfied: $\mathbf{v}'_i + \delta_i \in guard(e'_i)$. The transition uniquely determines the next state $(q'_{i+1}, \mathbf{v}'_{i+1})$, by the usual TA transition rule.

In the latter case, $(q'_i, \mathbf{v}'_i + \delta_i) \xrightarrow{\epsilon} (q'_{i+1}, \mathbf{v}'_{i+1})$ does not denote an actual transition, but the fact that the automaton A' remains in the same state: $q'_{i+1} = q'_i$ and $\mathbf{v}'_{i+1} = \mathbf{v}'_i + \delta_i$. The ϵ-transition uniquely determines the next state $(q'_{i+1}, \mathbf{v}'_{i+1})$, and its validity is ensured by definition.

Note that because of the ϵ-transitions, the run $\sigma(\rho)$ is not in a canonical form. Thus, an infinite run ρ may correspond to a finite run $\sigma(\rho)$ if $\sigma(\rho)$ is brought to canonical form. This case corresponds to the situation in which the property automaton does not make any more discrete transitions (different from ϵ) beyond a certain point.

The above arguments imply the following property:

Lemma 7.1 *σ associates an unique and valid run $\sigma(\rho)$ of A' to every run ρ of A.*

For a TPA $B = (A', F)$ and a correspondence function σ, let

$$\text{Exec}_{A,\sigma}(B) = \{\rho' \text{ run of } A' \mid \exists \rho \text{ run of } A,\ \rho' = \sigma(\rho)\}$$

We define also the following subset of $\text{Exec}_{A,\sigma}(B)$:

$$\text{Exec-inf}_{A,\sigma}(B) = \{\rho' \text{ run of } A' \mid \exists \rho \text{ infinite run of } A,\ \rho' = \sigma(\rho)\}$$

Satisfaction relations

We define two satisfaction relations for TPA:

- With the *safety* satisfaction relation ($\models_S$), a TA A satisfies a TPA $B = (A', F)$ (denoted $A \models_S B$) iff there is no run of A' in $\text{Exec}_{A,\sigma}(B)$ passing through a state from F.

  Intuitively, the states from F are regarded as *error* states, in which the property automaton must not enter.

- With the *liveness* satisfaction relation ($\models_L$), a TA A satisfies a TPA $B = (A', F)$ (denoted $A \models_L B$) iff every infinite run ρ of A corresponds to a run $\sigma(\rho)$ which brought to the canonical form is either

  1. *infinite*, and passes an infinite number of times through a state from F (Büchi acceptance condition), or
  2. *finite*, and ends in a state from F.

  Intuitively, the states from F are regarded as *progress* states, and every infinite run of A must make the property automaton B progress an infinite number of times.

The following notations are used in the definition of $\models_S$ and $\models_L$: let A be a TA and $B = (A', F)$ be a TPA with the components denoted in usual way. Let ρ be a run of A' in the canonical form. We define $\mathbf{inf}(\rho)$ as the set of discrete states q through which ρ passes an infinite number of times, if ρ is infinite, and the singleton formed of the last state of ρ, if ρ is finite. Let:

$$\text{Exec}^S_{A,\sigma}(B) = \{\rho \in \text{Exec}_{A,\sigma}(B) \mid \exists i \in \mathbb{N}.\, \mathbf{discrete}(\rho(i)) \in F\}$$
$$\text{Exec-inf}^L_{A,\sigma}(B) = \{\rho \in \text{Exec-inf}_{A,\sigma}(B) \mid \mathbf{inf}(\rho) \cap F = \emptyset\}$$

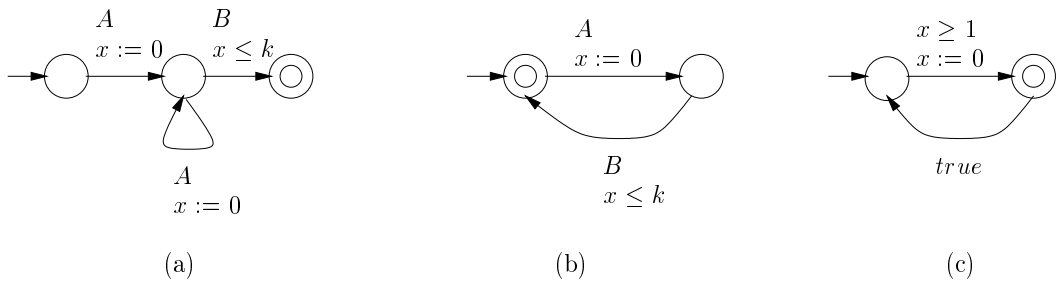


Figure 7.1: Examples of Timed Property Automata.

Definition 7.2 (TPA safety satisfaction) $A \models_S B$ *iff* $Exec^S_{A,\sigma}(B) = \emptyset$.

Definition 7.3 (TPA liveness satisfaction) $A \models_L B$ *iff* $Exec\text{-}inf^L_{A,\sigma}(B) = \emptyset$.

By definition, $\models_S$ may be used to check safety properties of a model, by negation: in order to specify that a certain bad condition does not occur during the execution of a system, the modeler builds a TPA that enters an error state when the condition is met in a corresponding run of the model.

$\models_L$ is strictly more powerful than $\models_S$, and may specify properties which have both a safety part and a liveness part. In order to express a safety property in a liveness TPA $B = (A, F)$, it is sufficient to introduce a sink state q in the TPA such that $q \notin F$. Then, the TPA must be built such that each (finite) run ρ of the model that does not satisfy the safety property leads the TPA in state q. Since every finite run is the prefix of at least one infinite run (in the sense that time may progress to infinity along it, but not necessarily containing an infinity of discrete steps), and every infinite continuation of ρ will leave the TPA in state q (because q is sink), the TPA will not be *liveness* satisfied if a run ρ leading to q exists.

The reason for which we introduce the two relationships is a practical one: the verification of *safety* satisfaction is equivalent to the verification of a simple reachability property, while the verification of *liveness* satisfaction involves a more complex algorithm for searching non-progress loops in the state space of a model. We also think that making a clear distinction between the two categories helps the modeler in the specification of properties.

Examples of properties

A simple example of *safety* property is: "an event A should never be followed by an event B at less than k time units". The TPA that describes this property is shown in Fig. 7.1-a[1]. The correspondence function used in verification would have to ensure that TPA transitions labeled with A and B correspond respectively to model transitions on which events A and B occur.

The classical *bounded response* property is a liveness property: "every event A is eventually followed by an event B within at most k time units". The TPA corresponding to this property is shown in Fig. 7.1-b.

The bounded response property may be expressed, in a slightly modified form, as a safety property: after an occurrence of the event A, k time units never pass without the occurrence of

[1]For representing TPA graphically, we use the same conventions as for timed automata. Additionally, states from the final set (F) are represented with double circles, and the urgency is not marked on transitions as all of them are *lazy* by definition.

B. The difference between the liveness version of the bounded response property and the above safety version is that, with the latter, zeno runs on which neither B occurs, nor k time units pass, are not considered to break the property. An example of such a property is specified in GOAL, later in this chapter.

Another example of liveness property is non-zenoness, which states that there is no infinite run of the system along which time remains below a finite limit. The specification of this property in liveness TPA is shown in Fig. 7.1-c. The correspondence function for this TPA must relate every transition of the model with the transition labeled *true*. The idea is that on every infinite run of the system, if time remains below a finite limit then the TPA will eventually remain in the non-accepting state (on the left) forever, which will lead to the non-satisfaction of the property.

Comparison with Timed Büchi Automata

The definition of TPA above is different from both versions of Timed Büchi Automata defined in [Alu91] and [Tri98]. The first difference is the correspondence relationship, which in [Alu91] and [Tri98] is state-based instead of transition-based. Thus, in [Alu91] and [Tri98], a function $\sigma : Q' \to 2^Q$ associates to each discrete state of B a set of corresponding states from A. The correspondence of runs from A and B is also defined based on the states rather than on transitions.

A correspondence based on transitions can be strictly more fine grained than a correspondence based on states. In consequence, there are properties referring to the transitions of A, such as: "a transition e is always triggered t time units after a transition e'", which can be expressed using the TPA defined above and which, in general, may not be expressible using the TBA from [Alu91, Tri98]. Conversely, properties referring to the states of A, such as "a state q is exited at most t time units after it is entered" can be expressed in both TBA and TPA.

From the point of view of the satisfaction relations, it is difficult to compare TPA to TBA, as the definition of TPA is based on a notion of correspondence of runs such that each run of the model corresponds to a run of the TPA, while TBA satisfaction is based on language inclusion [Alu91] or intersection [Tri98] and the sets of runs of the model and of the TBA are incomparable in general. We also note that satisfaction of TPA is decidable, as shown in the next section, while satisfaction of TBA is decidable with the definition based on intersection [Tri98] and undecidable with the definition based on inclusion [Alu91].

7.1.3 TPA model checking

In this section we discuss the problem of deciding TPA satisfaction. As the satisfaction relations $\models_S$ and $\models_L$ are based on the set of runs of B corresponding to runs of A, $\text{Exec}_{A,\sigma}(B)$, and on its subset $\text{Exec-inf}_{A,\sigma}(B)$, we construct an automaton generating the runs $\text{Exec}_{A,\sigma}(B)$.

Characterization of satisfaction as an automata language problem

Definition 7.4 *(weak synchronized product) Let A be a TA, $B = (A', F)$ a TPA, and $\sigma : Q' \times E \to E' \cup \{\epsilon\}$ the correspondence function between the edges of A and A'. We define the* weak synchronized product *of A and A' as: $A \boxtimes_\sigma A' = (\Sigma \times (\{\epsilon\} \cup \Sigma'), \mathcal{X} \cup \mathcal{X}', Q \times Q', (q_0, q'_0), T)$, where T is the minimal set of transition edges defined by the following rules:*

1. *$\forall e = (q_1, \zeta, u, a, X, q_2) \in E$ and $\forall q'_1 \in Q'$ such that $\sigma(q'_1, e) = \epsilon$, then $((q_1, q'_1), \zeta, u, (a, \epsilon), X, (q_2, q'_1)) \in T$.*

2. $\forall e = (q_1, \zeta, u, a, X, q_2) \in E$ *and* $\forall q_1' \in Q'$ *such that* $\sigma(q_1', e) = (q_1', \zeta', lazy, a', X', q_2')$, *then* $((q_1, q_1'), \zeta \wedge \zeta', u, (a, a'), X \cup X', (q_2, q_2')) \in T$, *and* $((q_1, q_1'), \zeta \wedge \neg\zeta', u, (a, \epsilon), X, (q_2, q_1')) \in T$.

The weak synchronized product defined above is similar to the AND synchronized product defined in §5.3: the function $\sigma : Q' \times E \to E' \cup \{\epsilon\}$ induces a relation $\tau = \{(e, e') \in E \times E' \mid \exists q' \in Q'. \sigma(q', e) = e'\}$, which defines a synchronized product $A \otimes_\tau A'$ almost identical with the weak synchronized product $A \boxtimes_\sigma A'$. This assertion is based on the following observations:

- as all transitions $e' \in E'$ have the urgency $u' = lazy$ (see definition of TPA), we have $max(u, u') = u$, $\forall u$
- from the definition of the product, it can be seen that all transitions of the automaton A' are either synchronizing with transitions from A, or not taken into account in the product. The reverse is not necessarily true, i.e. transitions of A may execute without synchronizing with A'.

The difference between $A \otimes_\tau A'$ and $A \boxtimes_\sigma A'$ comes from the existence of transitions $((q_1, q_1'), \zeta \wedge \neg\zeta', u, (a, \epsilon), X, (q_2, q_1')) \in T$[2]. They allow the automaton A to take a transition $e = (q_1, \zeta, u, a, X, q_2)$ without synchronizing with $e' = \sigma(q', e)$ even if A' is in the state q', in case the guard ζ' of e' is not satisfied. Therefore, the above operator enforces synchronization of A with A' only when the latter is ready to accept it (whence the attribute "weak"), and thus it does not constrain the behavior of A.

The above argument implies the following property:

Lemma 7.2 *There is a one-to-one correspondence between the runs of A and the runs of $A \boxtimes_\sigma A'$.*

Notations. In the following, we will use the following notations: let $\mathbf{v} : \mathcal{X} \cup \mathcal{X}' \to \mathbb{R}$ a valuation of the clocks of $A \boxtimes_\sigma A'$. We denote $\mathbf{v}|_A$ and $\mathbf{v}|_{A'}$ the restriction of $\mathbf{v}$ to $\mathcal{X}$ and respectively $\mathcal{X}'$, which are valuations of A and A'.

Let $\rho = ((q_0, q_0'), \mathbf{v}_0) \xrightarrow{\delta_0} ((q_0, q_0'), \mathbf{v}_0 + \delta_0) \xrightarrow{e_0} ((q_1, q_1'), \mathbf{v}_1) \xrightarrow{\delta_1} ...$ a run of $A \boxtimes_\sigma A'$. We denote $\rho|_A = (q_0, \mathbf{v}_0|_A) \xrightarrow{\delta_0} (q_0, \mathbf{v}_0|_A + \delta_0) \xrightarrow{e_0|_A} (q_1, \mathbf{v}_1|_A) \xrightarrow{\delta_1} ...$ where $e_i|_A$ is the transition e of A from which the transition e_i of $A \boxtimes_\sigma A'$ was constructed.

We also denote $\rho|_{A'} = (q_0', \mathbf{v}_0|_{A'}) \xrightarrow{\delta_0} (q_0', \mathbf{v}_0|_{A'} + \delta_0) \xrightarrow{e_0|_{A'}} (q_1', \mathbf{v}_1|_{A'}) \xrightarrow{\delta_1} ...$, where $e_i|_{A'} = e'$ if the transition e_i of $A \boxtimes_\sigma A'$ is constructed by synchronization with e', and $e_i|_{A'} = \epsilon$ if the transition e_i is not constructed by synchronization.

Sketch of proof. We argue that for every run ρ of $A \boxtimes_\sigma A'$, $\rho|_A$ is a uniquely determined run of A, and conversely, for every run π of A there is a unique run ρ of $A \boxtimes_\sigma A'$ such that $\pi = \rho|_A$. The proof in both directions may be based on an induction argument, over the steps of the runs. ∎

With the notations introduced above, the following property can be easily proved, using the same induction argument as in the previous property:

[2]Note that the guard of this transition $(\zeta \wedge \neg\zeta')$ may be a non-convex polyhedron, and thus does not conform to the definition of TA. However, this does not add complexity to the model, as $\zeta \wedge \neg\zeta'$ is a finite union of convex polyhedra $\zeta_1 \vee ... \vee \zeta_k$ and so the aforementioned transition may be considered to represent a set of alternative transitions with the convex guards $\zeta_1, ..., \zeta_k$, which falls back in the class of TA defined in 5.3.

Lemma 7.3 *For every run π of A, the corresponding run ρ of $A \boxtimes_\sigma A'$ is such that $\rho|_A = \pi$, and $\rho|_{A'} = \sigma(\pi)$, where σ is the correspondence function between the runs of the TA A and the runs of the TPA A'.*

Using the above result, the sets $\text{Exec}_{A,\sigma}(B)$ and $\text{Exec-inf}_{A,\sigma}(B)$ from the previous section, on which are based the definitions of the satisfaction relations, can be characterized as follows:

$$\begin{array}{lcll} \text{Exec}_{A,\sigma}(B) & = & \{\rho|_{A'} & | \quad \rho \text{ is a run of } A \boxtimes_\sigma A'\} \\ \text{Exec-inf}_{A,\sigma}(B) & = & \{\rho|_{A'} & | \quad \rho \text{ is an infinite run of } A \boxtimes_\sigma A'\} \end{array}$$

The satisfaction relations can therefore be characterized as follows:

- $A \models_S B$ iff there is no state (q, q') reachable in $A \boxtimes_\sigma A'$ such that $q' \in F$.
- $A \models_L B$ iff all infinite runs of $A \boxtimes_\sigma A'$ pass an infinitely often through at least one state $q' \in F$.

Abstractions for the verification of TPA satisfaction

The characterization from the previous section reduces the problem of deciding TPA *safety* satisfaction to the reachability problem for the automaton $A \boxtimes_\sigma A'$. As such, TPA *safety* satisfaction may be verified by constructing either the *region graph* or the *simulation graph* (see §5.4) of $A \boxtimes_\sigma A'$, and checking that no state (q, q') with $q' \in F$ is reachable in the constructed graph.

TPA *liveness* satisfaction can be decided in a similar way, by checking that there are no cycles in the *region graph* or the *simulation graph* which do not pass through states (q, q') with $q' \in F$. Such non-progress cycles can be detected using Tarjan's algorithm for finding strongly connected components [Tar72] or variants of it [HPY96, CVWY92].

The above arguments imply the *decidability* of both *safety* and *liveness* TPA satisfaction.

7.2 MSC

An MSC specification defines a set of event traces ordered in time. The first problem in using MSC as a formal property specification language is the lack of a semantics comprising the new timing aspects introduced in MSC-2000. In §7.2.1 we discuss a method of characterizing formally the language of traces specified by an MSC, as the set of accepting runs of a timed automaton with Büchi acceptance conditions. Our characterization works for a subset of MSC-2000, for several reasons; firstly, the language of event traces defined by a High-level MSC is not regular, as previously discussed in §4.1.3. Some simplifying assumptions have to be made so that the language may be characterized using automata. Secondly, certain constructs for specifying timing in MSC-2000 are outside the expressivity limits of timed automata. Finally, some parts of the MSC language are not discussed here (e.g. use of data), as in this work we concentrate on the specification of timing constraints.

A second problem in using MSC as property specification language for formal verification is to define the satisfaction relationship between SDL models and MSC specifications. The relationship is not defined in the standard Z.120 [IT99a], as it is considered outside the scope of the language definition. In §7.2.2 we examine some of the alternative definitions which may be given to the satisfaction relationship.

Based on the timed automata semantics defined for MSC, in §7.2.3 we discuss a method for verifying automatically the satisfaction of MSCs with timing constraints. The verification method reduces MSC satisfaction to TPA *liveness* satisfaction.

Throughout this work, we consider only the mechanisms for expressing timing constraints already available in MSC-2000, and we do not propose extensions to the language.

7.2.1 A timed automata semantics for MSC

In order to verify any form of MSC satisfaction, the language of traces defined by an MSC specification has to be defined formally. This is the task of the formal semantics of MSC, which is a part of the language standard (Annex B of Z.120). At the time being, however, the formal semantics of MSC does not cover the timing aspects introduced in MSC-2000 and defines only the untimed trace language generated by an MSC specification.

To be able to apply timed automata model checking techniques to the verification of MSC satisfaction, we define the set of timed traces generated by an MSC as the language of runs generated by a timed automaton with Büchi acceptance condition. For that, we take as starting point the Petri-net semantics of MSC given by [GPR93] and described in §4.1.3.

An MSC specification not containing timing annotations specifies a set of traces in which the relative delays between events may have arbitrary values. The timing annotations used to constrain the relative delays between events are (see §4.1.5):

- *relative constraints*, which specify the distance in time between the occurrence of two events, by a lower and an upper bound.
- *absolute constraints*, which specify the moment of occurrence of an event, by a lower and an upper bound.

The time values used in such constraints may either be specified statically (with expressions involving only *constants*) or dynamically (using unconstrained expressions of type `Time`). In the latter case, the expressions may contain values resulted from *measurements*, which may be either relative delays between events, or the absolute occurrence time of an event. In Fig. 4.4 we have shown an example containing both measurements and constraints.

However, not all the constructs mentioned before can be expressed using timed automata constructs. For example, measuring the distance in time between two events, and using it later to constrain the occurrence delays of other events is equivalent to using a *stop clock* operation. This operation is beyond the expressivity limits of timed automata.

Additionally, absolute constraints expressed in real-world time (like GMT) are also allowed in MSC. Such constraints raise problems, because there is no corresponding notion of absolute time in timed automata. In turn, absolute constraints referring to a time scale in which the first event of the MSC occurs at time 0 may be handled as relative constraints with respect to the first event of the MSC.

For these reasons, the semantics described in the following takes into account only *relative timing constraints* expressed using *constants*.

The timed automaton whose set of runs is equal to the set of timed traces generated by an MSC M is obtained in the following steps:

1. Build the Petri-net PN corresponding to the MSC M from which timing annotations have been removed, as explained in [GPR93] (and in §4.1.3).
2. Annotate the transitions of PN with timed automata-like guards and clock operations, as follows:

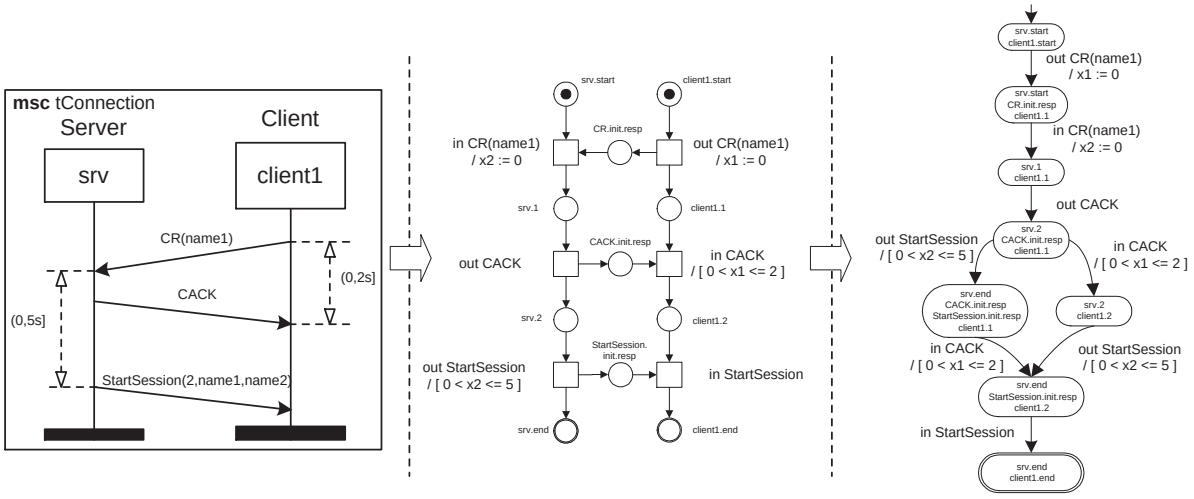


Figure 7.2: Semantics of timed MSC as timed automaton

- if the transition represents an MSC event which is the origin of a relative constraint C, annotate it with a clock reset "$x_c := 0$", where x_c is a unique clock corresponding to C.
- if the transition represents the finishing event of a constraint C, annotate it with a guard "$A \sim_1 x_c \sim_2 B$", where x_c is the clock corresponding to C, A and B are the lower and respectively upper bound of the constraint C, and $\sim_1, \sim_2 \in \{<, \leq\}$ according to whether the interval specifying C is open or closed in A, respectively B.

3. Build the LTS containing the reachable markings and transitions of PN. The definition of the timed automaton corresponding to an HMSC will only work if the graph of markings of PN is finite. However, as we have shown in §4.1.3, this condition is satisfied if it is considered that upper and lower boundaries of a Basic MSC constitute synchronization points for the instances contained in the MSC. In the following, we will consider that it is the case, as otherwise the MSC model checking problem is undecidable.
4. Annotate the transitions of the LTS with the same annotations (guards and clock resets) as the corresponding PN transitions. We obtain thus a timed automaton, in which all transitions have *lazy* urgency (by default).

In Fig. 7.2 we show the construction of the timed automaton corresponding to a simple MSC specification with timing annotations. First, in the center, the annotated Petri net is built. On the right we represented the timed automaton built from the LTS containing reachable markings of the Petri net. The states of the automaton are annotated with markings, that is with the labels of places in which there is a token.

For each MSC, there will be one or more markings of the Petri net corresponding to final states of the MSC. For the example in Fig. 7.2, the final marking is the marking in which the places srv.end and client1.end contain one token each. In the case of an HMSC, or a basic MSC using inline operators, there may be more than one final state of the MSC due to alternatives.

In the timed automaton, the final markings will generate one or more states which will be marked as final states. These states are usually sink states, but may also have outgoing

transitions if the MSC specification ends in a loop. For this reason, the timed automaton is defined to have Büchi acceptance conditions with respect to these final states. The set of timed traces described by an MSC may be defined as the language of runs generated by the timed Büchi automaton constructed above.

7.2.2 MSC satisfaction

In order to use MSC as a property specification language in relation to SDL models, the conditions in which an SDL model satisfies an MSC must be clearly defined. In the following, we discuss the principles of the definition of a satisfaction relationship, on the following axes:

1. the correspondence between MSC instances and SDL model components,
2. the correspondence between events specified in an MSC and events occurring in an SDL model,
3. the correspondence between MSC traces and SDL runs,
4. the relation between the set of traces defined by an MSC, and the set of runs of an SDL model.

Some of the ideas discussed in the sequel are inspired from the definition of MSC satisfaction used in the *Object*GEODE simulation and verification tool [TEL00a]. Nevertheless, the satisfaction relations considered in the tool are more restrictive and do not take into account the timing aspects.

SDL agents and MSC instances

If an MSC specification M describes a property of an SDL system S, then there must be a correspondence between instances of M and the distinct components of S to which they refer. As distinct components are modeled through *agents* in SDL, it is natural to assert that the instances of M should correspond to agents from S. This corresponds to the intended usage of MSC as explained in the standard.

Because agents are organized hierarchically, an instance of M designating an agent will also designate all its sub-agents. Events appearing on the instance correspond to events occurring in the agent itself or in the sub-agents. The environment (border) of M designates the environment of S

SDL events and MSC events

There is an intuitive correspondence between SDL events and MSC events, as the two languages were originally designed in order to be used jointly:

- *Message sending* corresponds to an SDL signal output. *Message receipt* may corresponds either to the receipt or to the consumption of a signal by an SDL agent. In the following, we consider that it corresponds to the consumption, i.e. to the execution of a matching **input** (or **priority input**) clause.
- A *timer operation* (set, reset) specified in an MSC corresponds to the execution of a matching timer operation in the SDL model. A *timer timeout* corresponds to the consumption of the matching *timer signal*.

- *Instance creations, instance stops, method calls* correspond to the analogous events in the SDL model.
- *Actions* and *conditions* are not matched against events in the SDL model.

SDL runs and MSC traces

For discussing the correspondence between SDL runs and MSC traces, we will consider the timed automata-based semantics of SDL discussed in §6.4. Thus, a run is a sequence of states and transitions: $\rho = q_0^\tau \xrightarrow{\delta_0} q_0^\tau + \delta_0 \xrightarrow{t_0} q_1^\tau \xrightarrow{\delta_1} q_1^\tau + \delta_1 \xrightarrow{t_2} q_2^\tau \ldots$, where $t_0, t_1, ...$ denote *discrete* transitions (either SDL transitions, or implicit *signal delivery* or *timeout* transitions) and $\delta_0, \delta_1, ...$ denote *time* transitions.

As specified by the timed semantics of MSC, an MSC trace is a sequence of discrete events separated by relative time durations $\psi = (e_0, \delta_0', e_1, \delta_1', e_2, \delta_2', ...)$. The correspondence between ρ and ψ depends on two aspects:

1. the correspondence between the events generated by discrete transitions $t_0, t_1, ...$ from ρ, and the events of ψ, $(e_0, e_1, ...)$. The correspondence may not be one-to-one, as there may be transitions generating 0 or multiple events.
2. the correspondence between the relative delays $\delta_0, \delta_1, ...$ from ρ and the relative delays $\delta_0', \delta_1', ...$ from ψ. Again, the correspondence may not be one-to-one, and depends on the correspondence between transitions and events. For example, if several events $e_i, ..., e_j$ are generated by the same transition t_k, the delays $\delta_i, ..., \delta_{j-1}$ must be 0.

We can define the correspondence between sequences of discrete events generated during an SDL model run, and (untimed) traces of events specified by an MSC, in several ways:

- If we consider that the MSC defines *complete* traces, then a run ρ of the SDL model is *inscribed* in the MSC if there is a trace ψ of the MSC which contains exactly the discrete events from ρ, in the same order.

  Alternatively, if we consider that the MSC defines *complete* traces with respect to a set of observable events E, then ρ is *inscribed* in the MSC if, by removing from ρ the events that are not in the observable set E, there is a trace ψ of the MSC which contains exactly the remaining events of ρ, in the same order.
- If we consider that the MSC defines *incomplete* traces, then ρ is *inscribed* in the MSC if there is a trace ψ of the MSC such that all the events of ψ appear in ρ in the same order, but ρ may contain additional events not appearing in ψ.

Relation between sets of runs and sets of traces

Globally, the satisfaction relation between an SDL model and an MSC specification may be defined in several ways, such as:

1. the SDL model S satisfies the MSC specification M ($S \vDash M$) if all runs of S are inscribed in M, or
2. $S \vDash M$ if there is a run of S which is inscribed in M, or
3. $S \vDash M$ if no run of S is inscribed in M.

Depending on the exact relation that is considered, an MSC may represent either a safety property, or a combined safety and a liveness property. For example, in case of the third relation described above, the MSC represents a purely safety property (the events in the MSC must not occur). In cases 1 and 2, the MSC has both a safety and a liveness part: the safety part is that events *not specified* in the MSC *must not occur*, and the liveness is that, *eventually*, all the events specified in the MSC must occur. For this reason, in general an MSC may be interpreted as a *liveness* timed property automaton.

7.2.3 Timed MSC model checking

The correspondence between MSC events and events occurring in the SDL model, described informally in the previous section, may be formalized as a correspondence function of the kind used in timed property automata (§7.1.2). This allows the interpretation of the timed automaton of an MSC as a TPA referring to the timed automaton of an SDL model. Thus, the verification of MSC satisfaction may in principle be performed using the model checking techniques described for TPA.

However, in the previous paragraph we showed that several kinds of MSC satisfaction relations may be useful. The verification of each kind of relation requires small modifications of the automaton corresponding to the MSC, discussed below.

We examine here the model checking method for one particular kind of MSC satisfaction relation, the methods for other types of relations being somewhat similar. We consider a relation in which MSCs represent *complete* event traces (that is, other events except those specified by the MSC are *not* allowed to occur in runs complying to the MSC). Moreover, by this relation, an SDL model is considered to satisfy an MSC if all the runs of the SDL model are either inscribed in the MSC (in the sense discussed in the previous section), or do not contain the events that appear in the beginning of the MSC (that is, they leave the TPA corresponding to the MSC in the initial state).

In order to verify this satisfaction relation, we have to modify the TPA corresponding to the MSC to ensure that traces containing additional observable events not appearing in the original MSC are *rejected*. Thus, in every state of the timed automaton, several additional transitions towards a new sink state (unexpected) are introduced. The transitions correspond to all observable events that are not expected in that state (or are expected with a different timing condition), according to the MSC specification. For example, in Fig. 7.3 we consider the state "srv.2,CACK.init.resp,client1.1" of the automaton from Fig. 7.2. Assuming that the set of observable events is formed only of the events appearing in the MSC, i.e. inputs and outputs of the signals CR,CACK and StartSession, the transitions that are added in the state are represented in the Fig. 7.3.

Another modification of the automaton concerns the initial state, which is also marked as final state. Thus, if during the synchronous execution of the SDL model and the MSC automaton, the MSC remains perpetually in the initial state along a run, that run will be considered compliant to the MSC (according to the definition of the Büchi acceptance condition of TPA).

With these modifications of the automaton corresponding to the MSC, the satisfaction of the MSC by an SDL model may be verified using the method for checking *TPA liveness satisfaction*. (We note however that small modifications of the algorithm for constructing the weak synchronized product between an SDL specification and an MSC property automaton are necessary, in order to accommodate the fact that one SDL transition may generate several events, and therefore it may trigger several transitions in the MSC automaton.)

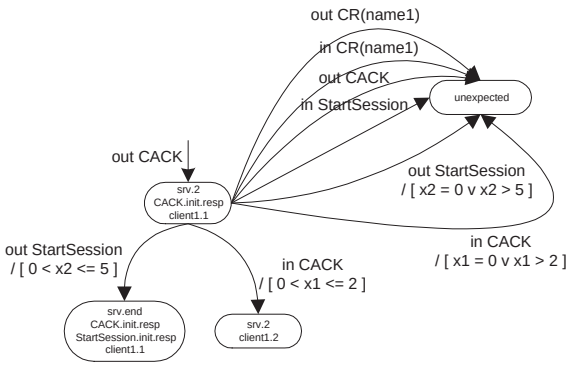


Figure 7.3: Augmenting the timed automaton for ensuring completion of traces

Other variants of MSC satisfaction may be verified similarly, by using a different method of augmenting the automaton (e.g. if the MSC specification is regarded as an incomplete trace, then unexpected events should not lead to the sink state). Throughout this work, we have only used the satisfaction relation described above, and we do not discuss further the other types of relations.

7.3 GOAL

GOAL observers, in the form defined in Chapter 4, cannot be used for the specification and verification of quantitative timing properties because they lack constructs for observing the timing of events. As we noted in §4.2.3, the value of the global SDL clock **now** may be tested by an observer, but due to the analysis method used by the *Object*GEODE tool this cannot be used in formal verification.

In consequence, in this section we examine a series of extensions to the GOAL language which enable the verification of properties involving time. The extensions are similar to those made for SDL, and introduce primitives taken from timed automata. The resulted language may be given a semantics in terms of timed property automata, defined in the beginning of this chapter.

7.3.1 Extensions for specification of timing constraints

The execution model of GOAL and the satisfaction relation between SDL models and GOAL observers are similar to those of timed property automata. Therefore, the GOAL extensions for observing timing proposed in this section are based on the mechanisms used in TPA – clocks and transition guards involving conditions on clocks.

Thus, an extended GOAL observer may declare explicit clocks, using the `Clock` data type introduced in SDL. The transition code of an observer may contain statements involving clocks, like in SDL: clock creation (using `mkClock`), reset (using `resetClock`) and assignments of clock variables.

The transition clauses of an extended observer may be guarded (using the **provided** clause) with clock constraints, of the forms allowed in SDL and in timed automata. The constraints may test the value of clocks of the observer or that of clocks belonging to the SDL model.

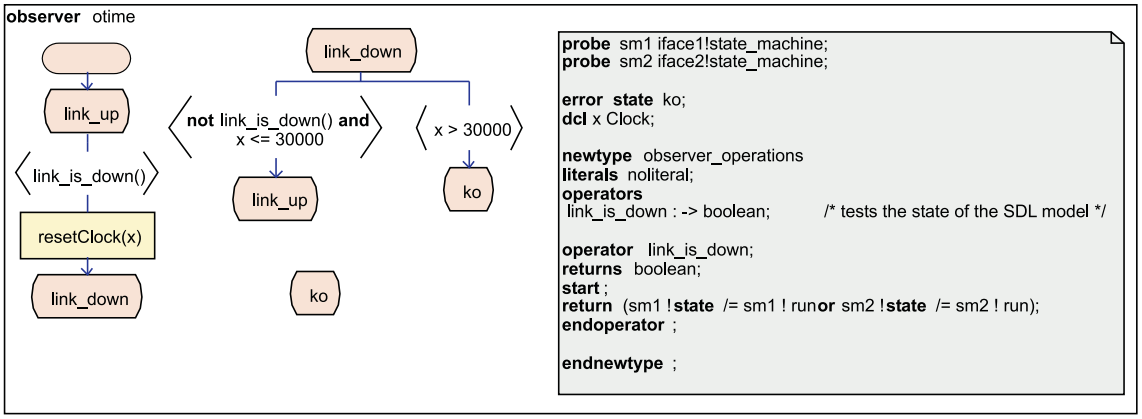


Figure 7.4: Timed safety property expressed in GOAL

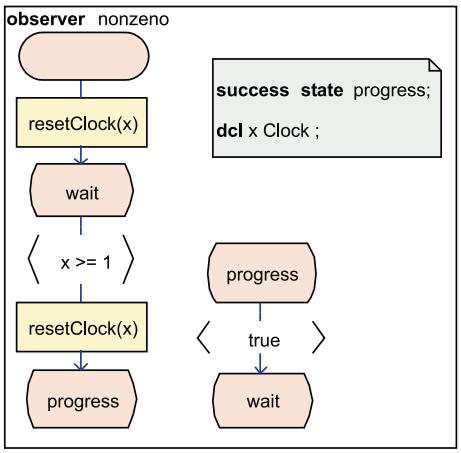


Figure 7.5: GOAL observer for checking non-zenoness in liveness mode

Like in TPA, urgencies are not allowed in GOAL and all transitions are considered *lazy*. The purpose of this restriction is to disallow observers to block time progress in the product automaton, as this would cut out valid behaviors of the SDL model from the state space of the product.

Fig. 7.4 contains an example of timed safety property of SpaceWire links (see the SpaceWire example in §6.2.4) expressed in GOAL with the extensions introduced above. The property may be expressed as follows: after a reset or a fault, a SpaceWire link is re-established in at most $30\mu s$[3]. We specify this property as a safety observer based on the assumption that the SpaceWire model does not allow zeno runs (see the discussion on *bounded response* properties at page 128).

Fig. 7.5 shows an observer which can be used to detect *zeno* runs in liveness mode: on each zeno run, the observer will eventually remain in the `wait` state forever, which will trigger the non-satisfaction of the observer. This observer is a direct transcription of the TPA in Fig. 7.1-c.

[3]This property should hold in the SpaceWire model, provided that the physical link is not damaged and the link is not reset a second time. These provisions may be ensured by other means in the *Object*GEODE verification tool (e.g. using transition filters).

7.3.2 Semantics and model checking of timed observers

The semantics of extended GOAL observers may be defined in terms of TPA. We do not intend to provide a complete definition of the semantics of extended GOAL observers here, but rather to outline the principles of this definition:

- The mapping of global states and transitions of a GOAL observer into timed automata states/transitions is similar to that explained in §6.4 in the case of SDL.
- The correspondence between events occurring in the SDL model, and events specified in GOAL transition clauses, which is at the basis of the execution model of GOAL (in the standard version), may be formalized as a correspondence function of the kind used in the definition of TPA. Thus, the correspondence function $\sigma : Q' \times E \rightarrow E' \cup \{\epsilon\}$ for GOAL observers may be defined as follows:
  - If a transition t' from the TPA of the observer, starting in a state q', is triggered by a **when** clause[4], then for every transitions t from the TA of the SDL system producing the observed event, we have $\sigma(q', t) = t'$. By definition, GOAL observers must be deterministic, which means that there is at most one t' starting in q' and satisfying the above conditions.
  - If a transition t' from the TPA of the observer, staring in a state q', is triggered by a **provided** clause[5], then for every transitions t from the TA of the SDL system which leads to a global SDL state in which the **provided** condition holds, we have $\sigma(q', t) = t'$.
  - For all other pairs of SDL transitions t and GOAL states q', we have $\sigma(q', t) = \epsilon$.
- The satisfaction relation between SDL models and GOAL observers is similar to the satisfaction relations defined for TPA. Thus, properties are specified in GOAL by annotating states as *error* or *success* states. The *safety* and *liveness* verification of GOAL satisfaction (as described in §4.2.2) correspond respectively to the verification of *safety* and *liveness* satisfaction relations for TPA. The verification is based on building a product between the SDL model's state space and the GOAL observer's state space, which is similar to the weak synchronized product defined for TPA:
  - GOAL transitions always synchronize with an SDL transition; the reverse is not true.
  - Synchronization is based on the correspondence relationship between SDL transitions and GOAL transitions.
  - Synchronization with a GOAL observer does not restrain the set of behaviors of the SDL model (synchronization is *weak*).

As in the case of MSC, a transition in the SDL model may produce several observable events. Taking them into account requires small modifications of the definition of the weak synchronized product, so that the observer may take several steps with a single step of the SDL model.

With the semantics of GOAL observers defined in terms of timed property automata, the satisfaction of an observer by an SDL model may be verified using the techniques applicable to

[4] **When** is used for observing discrete events in the SDL model (outputs, inputs, transitions firing, etc.). See the definition of GOAL in §4.2 and [TEL00a].

[5] **Provided** is used for observing elements of the state of the SDL model (variables, queues, etc.). See the definition of GOAL in §4.2 and [TEL00a].

TPA, discussed in §7.1.3. The tool presented in the next chapter uses the *simulation graph* of the weak synchronized product between the observer TPA and the SDL model TA, as abstraction for verifying observer satisfaction. However, neither the SDL model TA nor the observer TPA are built explicitly by the tool; instead, the tool builds directly the simulation graph of the product and checks properties (reachability and liveness) on the fly.

7.4 Discussion

We have approached the problem of specifying and verifying quantitative temporal properties of SDL models using the MSC and GOAL languages. Our study shows that a few timing-related extensions make GOAL into an expressive and flexible language for specifying properties. On the other hand, MSC proves to be less flexible for specifying properties. Even if timing-related extensions are not needed a priori because MSC-2000 contains constructs for specifying timing, the constraints expressible in MSC (based on time intervals) may be less powerful than the explicit-clock approach adopted in GOAL. Moreover, we have shown that several definitions for the notion of MSC satisfaction are possible and equally justified, and extensions of the language would be needed in order to let the user specify the exact meaning of satisfaction.

Related work

As GOAL is a proprietary language, there is no previous work on the specification of timing properties in it. However, the language is by nature related to other operational property specification languages such as (timed) Büchi automata [Alu91], from which we have taken our inspiration in defining the extensions.

On the side of MSC, even if much research has been dedicated to the analysis of MSC specifications themselves, there are few results on using MSC as a property language in relation to other system description formalisms. A notable approach is represented by the Live Sequence Charts, an MSC variant proposed in [DH98]. LSCs provide a definition for the notion of satisfaction based on a set of language extensions, which basically allow the distinction between possible and necessary behavior, but also include facilities like activation conditions for portions of an MSC. The extensions introduced by LSC solve some of the problems with MSC pointed out in this work. However, [DH98] does not take into consideration timing issues in the definition of LSC semantics.

Chapter 8

Timed SDL simulation and verification

In this chapter we discuss a simulation and verification tool built in the context of this work, which is based on SDL, MSC and GOAL, and implements the language extensions described in the previous chapters. Our tool is based on an existing industrial SDL environment, *Object*GEODE [TEL00a], which provides many functionalities (graphical editing of SDL, MSC and GOAL, simulation, verification, automated test generation, executable code generation and others) and implements most of the features of SDL-96, MSC-96 and GOAL. The advantage of reusing parts of an industrial environment is that the implementation of most of the features of the three languages, which are not affected by the proposed timing extensions, is obtained without additional efforts.

We begin the chapter by a high-level description of the tool architecture and main functionalities, in §8.1. In §8.2 we describe the construction of the timed simulation graph, the central function of our tool on which all other features are based. Finally, §8.3 examines the tool from a user point of view, discussing specific commands and functionalities.

8.1 Tool architecture and functioning

The verification tool built in the context of this work reuses the overall architecture and the main functions of the *Object*GEODE simulation and verification tool (*Simulator*). *Object*GEODE is a CASE[1] environment which assists the software designer in a number of system development tasks, ranging from analysis and design, to validation, code generation, test generation and documentation. It supports several types of models, including the SDL, MSC and GOAL languages discussed previously, as well as certain types of UML diagrams [OMG99]. A complete description of the environment is given in [TEL00a] and on the tool website[2].

In this context, the Simulator provides the following functions:

- Simulation of SDL models, with debugging features similar to those offered in most modern programming environments (step-by-step execution, (conditional) breakpoints, data watch, etc.). Some advanced debugging features are available:
  - reverse execution (undo/redo),

[1] Computer Aided Software Engineering
[2] http://www.telelogic.com/ObjectGeode

  - scenario archiving and rerun,
  - automatic stimulation of open models with signals,
  - automatic guidance through transition filters,
  - production of customized graphical traces in MSC,
  - model coverage analysis,
  - interactive or random scheduling of fired transitions.

- Verification functions, using an exhaustive exploration of the model state space:
  - deadlock detection,
  - configurable dynamic error detection (e.g. lost or unexpected signals),
  - checking satisfaction of properties written in MSC or GOAL.

Both simulation and verification functions are based on the construction of the state space (simulation graph) of the model. They differ in the order in which this state space is constructed. In simulation, only the states along the executed scenario (which may be guided interactively by the user, or constructed at random) are built, while in verification, a larger part of the state space is constructed (in depth-first or breadth-first order). Verification of properties is done on the fly and the entire state space needs to be built only in some cases.

The exploration of the state space relies on a representation of the global state of the SDL model (and the associated MSC and GOAL observers), comprising the values of all variables in the model (and in observers), the discrete states of all agents (and observers), and the contents of signal queues. The successors of a state are computed in several steps:

1. Evaluation of SDL transitions which are fireable in the current state of the model, and selection of a transition to be fired (depending on the execution mode: interactive, random, verification). In the commercial version of the tool, the list of fireable transitions may contain either explicit discrete transitions (SDL model transitions), implicit discrete transitions (timer expiration, signal discard, etc.), or time transitions. In the extended version of the tool presented here, time transitions are handled implicitly, as an abstraction similar to the timed automata simulation graph is used; therefore, only discrete transitions (explicit or implicit) appear in this list.
2. Execution of the actions specified by the transition, on the current state of the SDL model. A new state of the SDL model is thus obtained. The observable events occurring during the transition are recorded and used in the next step.
3. Execution of transitions in the associated MSC and GOAL observers, triggered by the events recorded in the previous step. A new state for each MSC and GOAL observer is thus obtained. These are merged with the state of the SDL model, to obtain a new global state.

Other actions may be executed in parallel with these steps, to accomplish different functionalities of the Simulator: transition filtering (during the first step), test for stop conditions (after step 3), update of code coverage tables (during step 2), etc.

The software components involved in the simulation of a model are shown in Fig. 8.1. The SDL model and the MSC and GOAL properties are compiled in an executable format, in which SDL transitions for example are transformed into routines that rely on a set of action primitives (corresponding to SDL actions) implemented in a model-independent library (*Simulator library*

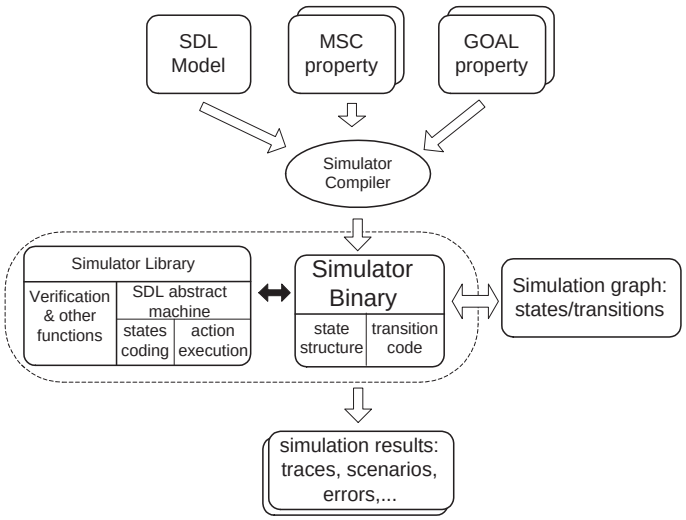


Figure 8.1: Simulation tool architecture

in Fig. 8.1). This library also contains the definitions of the generic data structures used for coding global model states, as well as the implementation of the auxiliary functionalities of the Simulator (such as verification procedures). Furthermore, the simulator binary interacts with a user interface component, through which the user guides the simulation session and obtains the results.

As noted before, the central function of the simulator is the construction of the state space. Implementing the language extensions and the verification techniques described in the previous chapters will impact both the data representation of the state space and the construction procedures. The next section examines these aspects.

8.2 The timed simulation graph

The tool built in the context of this work uses an abstraction similar to the *simulation graph* of timed automata for handling the clocks of the SDL model (and of the MSC or GOAL observers). The states manipulated by the extended simulator are symbolic states of the form (q, S), where q is a global state of the SDL system identical to the global states manipulated by the standard simulator. S is a zone of the clock space (see §5.4), representing the set of clock valuations reachable by the scenario executed so far (i.e. the path from the initial state to the current state, in the state space). Thus, a simulation state (q, S) represents a set of explicit model states $(q, \mathbf{v})$ which are reachable from the initial state by a same sequence of discrete transitions, and which share the same discrete part.

Transitions in this simulation graph correspond only to discrete transitions of the SDL model (i.e. there are no time transitions). The successor of a simulation state (q, S) after execution of a discrete transition e, is computed by the extended Simulator as $\mathsf{time\text{-}succ}(\mathsf{disc\text{-}succ}(e, (q, S)))$, where the operators time-succ and disc-succ were defined for timed automata in §5.4.

In the following, we discuss the data representation of simulation states, and the effective method for computing the successors of a state.

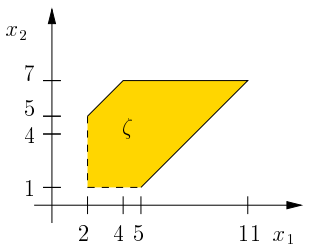


Figure 8.2: Bi-dimensional clock zone

8.2.1 Representation of states

For encoding the discrete part of a simulation state (q, S), the same representation as in the standard version of the simulator is used. For representing clock zones, we use the *difference bounds matrices* (DBM) proposed in [Dil89, ACD93].

A DBM is a square matrix, which can encode a conjunction of atomic clock conditions (convex polyhedron in the clock space). A DBM with $(n+1)^2$ elements is used for encoding a polyhedron in an n-dimensional clock space. The elements of a DBM are pairs of the form $(c, \sim)$, with $c \in \mathbb{Z} \cup \{\infty\}$ and $\sim \in \{<, \leq\}$.

Let $\{x_1, ..., x_n\}$ be the set of clock over which a DBM M is defined, and $M(i,j)$ denote the element of M at coordinates i, j. We assume that the rows and columns of M are numbered from 0 to n. For all $i \neq 0$ and $j \neq 0$, $M(i,j)$ encodes the upper constraint on the clock difference $x_i - x_j$: if $M(i,j) = (c, \sim)$ then $x_i - x_j \sim c$. Column 0 and row 0 are used to encode the constraints of individual clocks: if $M(i,0) = (c, \sim)$ and $M(0,i) = (c', \sim')$ then $-c' \sim' x_i \sim c$.

Take for example the zone ζ in a bi-dimensional clock space, given by the following constraints: $x_1 > 2$, $1 < x_2 \leq 7$ and $-4 \leq x_2 - x_1 \leq 3$. The zone is represented in Fig. 8.2. The DBM that encodes this zone is:

		x_1	x_2
	$(0, \leq)$	$(-2, <)$	$(-1, <)$
x_1	$(11, \leq)$	$(0, \leq)$	$(4, \leq)$
x_2	$(7, \leq)$	$(3, \leq)$	$(0, \leq)$

We note that a same convex polyhedron may be represented by several DBMs, if some of the comparisons are useless. In the above example, the inequality $x_1 le 11$ may be inferred from $x_2 \leq 7$ and $x_1 - x_2 \leq 4$. Any DBM representing the latter inequalities, and an additional inequality $x_1 \leq c$ with $c \geq 11$ will represent correctly the same polyhedron ζ, regardless of the actual value of c. This is because the intersection of the conditions represented in all such DBMs is the same.

However, a canonical form for DBMs may be defined, such that every convex polyhedron is represented by a unique DBM. Such a canonical form, and an algorithm for bringing an arbitrary DBM to the canonical form are presented in [Tri98]. The canonical form is important, as its existence simplifies the test of equality for polyhedra (which is reduced to plain equality of DBMs).

All operations on convex polyhedra needed for the construction of a timed automata simulation graph (intersection, test for inclusion, projections, etc.) can be easily implemented using DBMs. For this reason, many tools including KRONOS [Yov97, DOTY95], UPPAAL [LPY97, BLL+96] and IF [BFG+99, Boz99] use this data structure.

The drawback of DBMs is they cannot represent non-convex polyhedra, which arise in the analysis of timed automata with urgency, and consequently in the analysis of extended SDL specifications. As non-convex polyhedra may be represented with unions of convex polyhedra, they are manipulated in the simulator using lists of DBMs. This representation is not canonical, which complicates the implementation of the equality test for non-convex polyhedra, and penalizes the performance of the tool in certain cases.

8.2.2 Transition steps

In this section we present the algorithm used to construct successors of a simulation state (q, S). For the beginning, we will consider the case of an SDL model which is simulated without an associated MSC or GOAL observer.

Auxiliary polyhedra operations

The computation of successors of a simulation state equates to the computation of the list of enabled transitions, followed by the computation of time-succ(disc-succ$(e,(q,S))$) for each enabled transition e. However, the definitions of time-succ and disc-succ (§5.4) are descriptive, and do not provide an operational procedure for computing these operations. In the following, we define several operations on clock polyhedra, which are used in addition to usual operations (intersection, union, complementation) in the computation of the time-succ and disc-succ operations. In all operations, we consider only clock valuations yielding positive values (in $\mathbb{R}_+$), as it is the case for all clocks appearing in extended SDL models.

1. *Orthogonal projections.* Let ζ be a polyhedron on the clock set $\mathcal{X}$, and $y \in \mathcal{X}$ a clock. The orthogonal projection of $\mathcal{X}$ parallel with the axis of y is the polyhedron

$$\zeta[y := 0] = \left\{ \mathbf{v} \in \mathbb{R}_+^{\mathcal{X}} \mid \mathbf{v}(y) = 0 \wedge (\exists \mathbf{v}' \in \zeta.\, \forall x \in \mathcal{X} \setminus \{y\}.\ \mathbf{v}(x) = \mathbf{v}'(x)) \right\}$$

   The operation is used to implement clock reset in the construction of the simulation graph. Fig. 8.3-b shows the result of the application of this operation on a polyhedron in a 2-clock space (represented in Fig. 8.3-a):

$$\begin{array}{rcl} \zeta & = & \{\, \mathbf{v} \in \mathbb{R}_+^{\{x,y\}} \mid 2 \le \mathbf{v}(x) \le 8 \ \wedge\ 1 \le \mathbf{v}(y) \le 7 \ \wedge\ -4 \le \mathbf{v}(y) - \mathbf{v}(x) \le 3 \,\} \\ \zeta[y := 0] & = & \{\, \mathbf{v} \in \mathbb{R}_+^{\{x,y\}} \mid 2 \le \mathbf{v}(x) \le 8 \ \wedge\ \mathbf{v}(y) = 0 \,\} \end{array}$$

2. *Forward diagonal projection.* The forward diagonal projection of a polyhedron ζ gives the clock valuations reachable from valuations of ζ by letting time pass with whatever amount:

$$\nearrow \zeta = \left\{ \mathbf{v} \in \mathbb{R}_+^{\mathcal{X}} \mid \exists \mathbf{v}' \in \zeta.\, \exists \delta \in \mathbb{R}_+.\, \forall x \in \mathcal{X}.\ \mathbf{v}(x) = \mathbf{v}'(x) + \delta \right\}$$

   The operation is used to implement time progress in the simulation graph. The result of $\nearrow \zeta$ for the polyhedron taken as example above is represented in Fig. 8.3-c:

$$\nearrow \zeta = \{\, \mathbf{v} \in \mathbb{R}_+^{\{x,y\}} \mid 2 \le \mathbf{v}(x) \ \wedge\ 1 \le \mathbf{v}(y) \ \wedge\ -4 \le \mathbf{v}(y) - \mathbf{v}(x) \le 3 \,\}$$

3. *Backward diagonal projection.* The backward diagonal projection of ζ contains the clock valuations from which the valuations of ζ are reachable by letting time pass with whatever amount:

$$\swarrow \zeta = \left\{ \mathbf{v} \in \mathbb{R}_+^{\mathcal{X}} \mid \exists \mathbf{v}' \in \zeta.\, \exists \delta \in \mathbb{R}_+.\, \forall x \in \mathcal{X}.\ \mathbf{v}(x) = \mathbf{v}'(x) - \delta \right\}$$

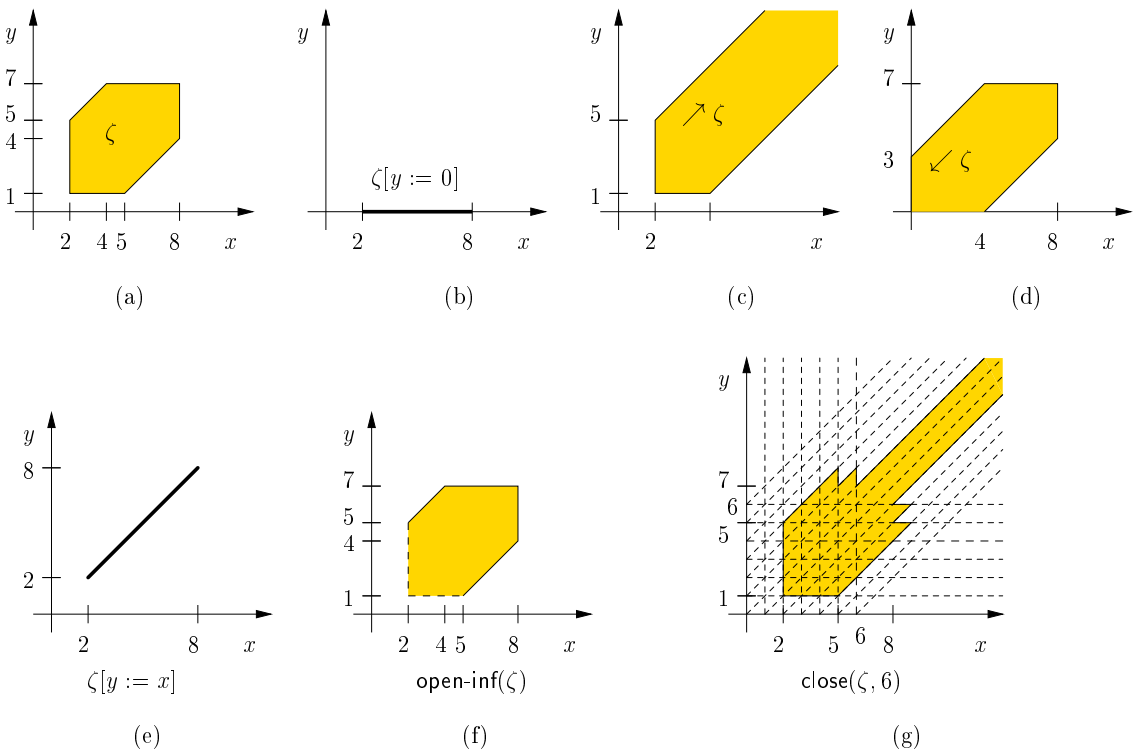


Figure 8.3: Operation on clock polyhedra

For our example, the result of this operation is (see Fig. 8.3-d):

$$\swarrow \zeta = \{\, \mathbf{v} \in \mathbb{R}_+^{\{x,y\}} \mid 0 \le \mathbf{v}(x) \le 8 \;\wedge\; 0 \le \mathbf{v}(y) \le 7 \;\wedge\; -4 \le \mathbf{v}(y) - \mathbf{v}(x) \le 3 \,\}$$

4. *Clock to clock assignment.* This operation is used to implement assignments between clocks in the simulation graph. For a polyhedron ζ, the result of an assignment $y := x$ is defined as follows:

$$\zeta[y := x] = \left\{\, \mathbf{v} \in \mathbb{R}_+^{\mathcal{X}} \mid \mathbf{v}(y) = \mathbf{v}(x) \wedge (\exists \mathbf{v}' \in \zeta.\, \forall x \in \mathcal{X} \setminus \{y\}.\; \mathbf{v}(x) = \mathbf{v}'(x)) \,\right\}$$

For our example, the result of this operation is (see Fig. 8.3-e):

$$\zeta[y := x] = \{\, \mathbf{v} \in \mathbb{R}_+^{\{x,y\}} \mid 2 \le \mathbf{v}(x) \le 8 \;\wedge\; 2 \le \mathbf{v}(y) \le 8 \;\wedge\; \mathbf{v}(y) = \mathbf{v}(x) \,\}$$

5. *Inferior opening.* This operation calculates the largest polyhedron $\zeta' \subseteq \zeta$ which is open, in the topological sense, on the faces determined by lower clock bounds:

$$\mathsf{open\text{-}inf}(\zeta) = \{\, \mathbf{v} \in \zeta \mid \exists \delta \in \mathbb{R}_+.\, \delta > 0 \wedge \mathbf{v} - \delta \in \zeta \,\}$$

Note that if all the lower clock bounds of a polyhedron ζ involve strict inequalities, then $\mathsf{open\text{-}inf}(\zeta) = \zeta$. In the above definition we use $\mathbf{v} - \delta$ to denote the valuation $\mathbf{v}'$ with $\mathbf{v}'(x) = \mathbf{v}(x) - \delta$ for all clocks x.

The result of the application of this operator on the example taken before is shown in Fig. 8.3-f.

6. *c-closure.* This operation is used to compute the closure of the polyhedron ζ with respect to the region equivalence relationship $\simeq_c$ defined in §5.4. The operation is used in order to render the number of regions (and zones) finite. The definition of c-closure with respect to a constant c is:

$$\mathsf{close}(\zeta, c) = \left\{ \mathbf{v} \in \mathbb{R}_+^{\mathcal{X}} \mid \exists \mathbf{v}' \in \zeta . \mathbf{v} \simeq_c \mathbf{v}' \right\}$$

An example of the partitioning of the clock space $\mathbb{R}_+^{\mathcal{X}}$ into equivalence classes with respect to $\simeq_c$ was given in Fig. 5.2. In Fig. 8.3-g, we represent the polyhedron $\mathsf{close}(\zeta, 6)$, as well as the lines that partition the space in equivalence classes (dotted). Note that the c-closure operation may yield a non-convex polyhedron even if the initial polyhedron is convex.

7. *Increasing/decreasing the dimension of the clock space.* These operations take as parameter a polyhedron in an n-dimensional space, and yield a polyhedron in a space of dimension $n+1$ or $n-1$.

Let $\zeta \in \mathbb{R}^{\mathcal{X}}$, and $x \notin \mathcal{X}$. We define[3]:

$$\zeta[\uparrow x] = \{ \mathbf{v} \in \mathbb{R}^{\mathcal{X} \cup \{x\}} \mid \mathbf{v}|_{\mathcal{X}} \in \zeta \wedge \mathbf{v}(x) = 0 \}$$

Let $\zeta \in \mathbb{R}^{\mathcal{X}}$, and $x \in \mathcal{X}$. We define:

$$\zeta[\downarrow x] = \{ \mathbf{v} \in \mathbb{R}^{\mathcal{X} \setminus \{x\}} \mid \exists \mathbf{v}' \in \zeta . \mathbf{v} = \mathbf{v}'|_{\mathcal{X} \setminus \{x\}} \}$$

All the above operations may be easily implemented using the DBM representation of polyhedra. A more detailed discussion on the properties of some of these operations, and their implementation using DBMs can be found in [Tri98]. Some additional operations are introduced here, compared to [Tri98]: the clock assignment, the inferior opening, and the dimension increasing/decreasing operators. Their implementation with DBMs does not pose difficulties.

Implementation of state operators

The state operators time-succ and disc-succ may be expressed as combinations of operations on polyhedra, taking simulation state zones and transition guards as operators. In this section we introduce the formulas for calculating time-succ and disc-succ, which are used by the simulation tool in the construction of the simulation graph.

We remind the definition of disc-succ for timed automata:

$$\mathsf{disc\text{-}succ}(e, (q, S)) = (q', \{\mathbf{v}' \mid \exists \mathbf{v} \in S . (q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')\})$$

where $e = (q, \zeta, u, a, X, q')$ is a transition between q and q'.

In the SDL simulator, the discrete destination state q' is obtained from the discrete source state q, by applying the rules of the dynamic semantics of SDL. However, for obtaining the zone $S' = \{\mathbf{v}' \mid \exists \mathbf{v} \in S . (q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')\}$ from the initial zone S, several polyhedra operations are used.

Let $X = \{x_1, ..., x_k\}$ be the clocks reset during the transition e. For timed automata as defined in Chapter 5, S' is given by:

$$S' = (S \cap \zeta)[x_1 := 0][x_2 := 0]...[x_k := 0]$$

[3] By $\mathbf{v}|_{\mathcal{C}}$ we denote the restriction of a function ($\mathbf{v}$) on a subset of its domain ($\mathcal{C}$)

The intersection $S \cap \zeta$ yields the valuations $\mathbf{v} \in S$ such that e is enabled in the automaton state $(q, \mathbf{v})$. The orthogonal projections following the intersection take into account clock resets.

This formula was previously described for timed automata without urgency in [Tri98]. It applies in the same way in our framework based on timed automata with urgency. For this reason we do not include here a proof of correctness. We note however that, in the simulation tool described in this work, the additional clock operations defined in SDL have been taken into account:

- *Clock assignments*, implemented using the clock assignment operator $[x := y]$
- *Clock creation and deletion*, implemented using the operators for increasing/descreasing the dimension of the clock space: $[\uparrow x]$ and $[\downarrow x]$.

In the following, we examine the computation of time-succ in timed automata with urgency, which is is slightly more complicated as it has to take into account the specific time progress conditions using urgencies. For simplicity and precision, we will use the timed automata notations instead of those specific to the SDL semantics. We remind the definition of time-succ:

$$\text{time-succ}((q, S)) = (q, \{\mathbf{v}' \mid \exists \mathbf{v} \in S,\ \delta \in \mathbb{R}.\ (q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}')\})$$

We rewrite the time progress conditions of timed automata with urgency, given in Chapter 5 on page 84, by making some variable changes: $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}')$ iff $\mathbf{v}' = \mathbf{v} + \delta$ and:

1. $\forall e = (q, \zeta, u, a, X, q')$ transition starting from q such that $u = \mathit{eager}$, $\forall \delta' \in (0, \delta]$, $\mathbf{v}' - \delta' \notin \zeta$.
2. $\forall e = (q, \zeta, u, a, X, q')$ transition starting from q such that $u = \mathit{delayable}$, $\forall \delta'$, δ'' such that $0 \leq \delta'' < \delta' \leq \delta$, $(\mathbf{v}' - \delta' \in \zeta \ \Rightarrow\ \mathbf{v}' - \delta'' \in \zeta)$.

Let S' denote the zone $\{\mathbf{v}' \mid \exists \mathbf{v} \in S,\ \delta \in \mathbb{R}.\ (q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}')\}$ from the definition of time-succ, which we want to characterize using the polyhedra operations defined previously. We will define the operators restrict-eager(S, ζ) and restrict-delayable(S, ζ), which yield the time successors of a zone S restricted by a time progress condition as imposed by one *eager*, and respectively *delayable* transition having the guard ζ. The idea is that S' is the intersection of these restrictions, restricted with the guards of all discrete transitions e leaving from the state q.

For obtaining the expression of restrict-eager(S, ζ), we consider the valuations $\mathbf{v} \in S$ and their possible time successors. Depending on the position of $\mathbf{v}$ in S, we have the following three cases:

1. if $\mathbf{v} \in \zeta$, then the *eager* transition with the guard ζ is enabled in $\mathbf{v}$, and therefore time may not progress at all from $\mathbf{v}$.

   The restricted time successors of $\mathbf{v}$ are: restrict-eager$(\{\mathbf{v}\}, \zeta) = \{\mathbf{v}\}$.

2. if $\mathbf{v} \in (\swarrow \zeta) \backslash \zeta$, then time may progress from $\mathbf{v}$ but there is a point at which the successors of $\mathbf{v}$ intersect the guard ζ, and from which time may no longer progress.

   The restricted time successors of $\mathbf{v}$ are: restrict-eager$(\{\mathbf{v}\}, \zeta) = (\nearrow \{\mathbf{v}\}) \cap ((\swarrow \zeta) \setminus$ open-inf$(\zeta))$.

3. if $\mathbf{v} \notin \swarrow \zeta$, then time may progress indefinitely from $\mathbf{v}$.

   The restricted time successors of $\mathbf{v}$ are: restrict-eager$(\{\mathbf{v}\}, \zeta) = \nearrow \{\mathbf{v}\}$.

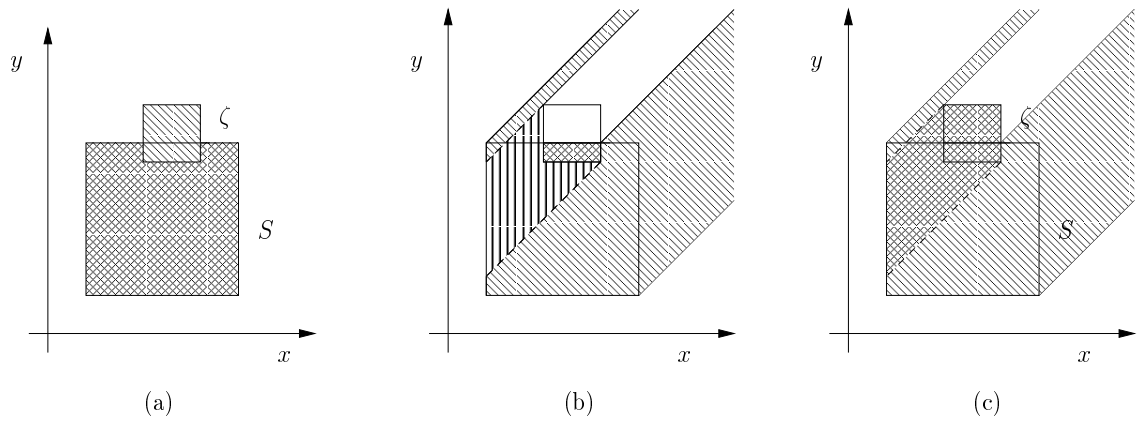


Figure 8.4: Urgency restriction operators.

The definition of restrict-eager(S, ζ) is then:

$$\begin{aligned}\text{restrict-eager}(S,\zeta) \quad = \quad & (S \cap \zeta) \cup \\ & (\nearrow (S \cap (\swarrow \zeta \setminus \zeta))) \cap (\swarrow \zeta \setminus \text{open-inf}(\zeta)) \cup \\ & (\nearrow (S \setminus \swarrow \zeta))\end{aligned}$$

In Fig. 8.4-b we show an example of application of the restrict-eager operator, for a square zone S and a square guard ζ (represented in Fig. 8.4-a). The zones that correspond to cases 1, 2 and 3 above are the crosshatched zone, the zone filled with vertical lines, and respectively the zone filled with oblique lines.

We proceed in the same manner for obtaining the expression of restrict-delayable(S, ζ). Let $\mathbf{v} \in S$.

1. if $\mathbf{v} \in \swarrow \zeta$, the there is a point at which the successors of $\mathbf{v}$ intersect the guard ζ. Time may progress as long as the successors remain in ζ.

   The restricted time successors of $\mathbf{v}$ are restrict-delayable$(\{\mathbf{v}\}, \zeta) = (\nearrow \{\mathbf{v}\}) \cap (\swarrow \zeta)$.

2. if $\mathbf{v} \notin \swarrow \zeta$, then time may progress indefinitely from $\mathbf{v}$.

   The restricted time successors of $\mathbf{v}$ are restrict-delayable$(\{\mathbf{v}\}, \zeta) = \nearrow \{\mathbf{v}\}$.

The definition of restrict-delayable for the entire zone S is then:

$$\begin{aligned}\text{restrict-delayable}(S,\zeta) \quad = \quad & (\nearrow (S \cap (\swarrow \zeta)) \cap (\swarrow \zeta)) \cup \\ & (\nearrow (S \setminus \swarrow \zeta))\end{aligned}$$

In Fig. 8.4-c we show an example of application of the restrict-delayable operator, for the same zone S and guard ζ as in the example before. The zones that correspond to cases 1 and 2 above are the crosshatched zone, and respectively the zone filled with oblique lines.
The characterization of time-succ$((q, S))$ is given by the following property:

Lemma 8.1 *Let q be a discrete state of a timed automaton, and $e_1, ..., e_l$ be the discrete transitions originating from s, with the guards (polyhedra) denoted respectively by $\zeta_1, ..., \zeta_l$.*

Then $\mathsf{time\text{-}succ}((q,S)) = (q,S')$, *where* S' *is given by:*

$$S' = \nearrow S \cap \left(\bigcap_{\substack{i\in\{1,\ldots,l\} \\ e_i \, is \, eager}} \mathsf{restrict\text{-}eager}(S,\zeta_i) \quad \cap \quad \bigcap_{\substack{i\in\{1,\ldots,l\} \\ e_i \, is \, delayable}} \mathsf{restrict\text{-}delayable}(S,\zeta_i) \right)$$

Proof. The proof can be found in Appendix B.

Successors computation algorithm

We describe below the algorithm used by the simulation tool to compute the successors of a simulation state (q,S). The algorithm is based on the semantic description of SDL given in §6.4, and on the successors computation formulas introduced in the previous section. It consists of the following steps:

1. *Evaluation of fireable transitions.* The transitions of the SDL model which fulfill the discrete conditions (i.e. not related to clocks) to be fired, are gathered in a list $t_1,...,t_n$. They depend only on the discrete part q of the state.

   Let $g_1,...,g_n$ be the clock guards of these transitions. A transition t_i is *enabled* in the symbolic state (q,S) if it is enabled in at least one explicit state $(q,\mathbf{v})$ contained in the symbolic state. This is tested by the condition $g_i \cap S \neq \emptyset$. Let $t'_1,...,t'_k$ be the transitions for which this condition holds (enabled transitions), and $g'_1,...,g'_k$ be their respective clock guards.

   In the symbolic state (q,S), certain explicit states $(q,\mathbf{v})$ may constitute deadlocks. They are the states in which no transition from $t'_1,...,t'_k$ is enabled, and in which no transition becomes enabled by letting time pass. The deadlocks form a zone characterized by the following expression: $S \setminus \bigcup_{j=1}^{k}(\swarrow g'_j)$. Therefore, in the construction of the simulation graph, if $S \setminus \bigcup_{j=1}^{k}(\swarrow g'_j) \neq \emptyset$ then the simulator will signal the existence of deadlocks.
2. *Transition firing.* For each fireable transition t'_i in the list computed before, the following steps are taken:
   (a) The state $(q',S') = \mathsf{disc\text{-}succ}(t'_i,(q,S))$ is computed. As noted in the previous section on page 147, this step consists of the following operations:
      i. The SDL statements specified by the transition t'_i are executed on the discrete part of the state (q,S), obtaining a new discrete state q'. S remains unchanged in this step.
      ii. $S'_0 = S \cap g'_i$ is computed. (q,S'_0) if the part of the symbolic state (q,S) on which the discrete transition t'_i is effectively enabled.
      iii. The clock operations (resets, assignments, creation, deletion) specified by the transition t'_i are successively executed on the zone S'_0. If o_1,o_2,...,o_k are the polyhedra operations corresponding to these SDL clock operations as explained on page 147, then the following polyhedra are successively computed: $S'_1 = o_1(S'_0),...,S'_k = o_k(S_{k-1})$.

   S' is the last computed polyhedron, S'_k.
   (b) The state $(q',S'') = \mathsf{time\text{-}succ}((q',S'))$ is computed. The computation uses the formula given in Lemma 8.1, and consists of the following steps:

i. $S_0'' = \nearrow S'$ is computed.

ii. The list of discrete, timeout, and signal delivery transitions enabled on a part of (q', S_0'') is obtained, in a step similar to Step 1. Let $t_1'', ..., t_l''$ be the enabled transitions, and $g_1'', ..., g_l''$ their clock guards (explicit guards for SDL transitions, implicit guards for *timeout* and *signal delivery* transitions, as explained in the timed semantics of SDL).

iii. For each *eager* transition or timeout transition t_i'' in the enabled list, S_0'' is intersected with $\mathsf{restrict\text{-}eager}(S', g_i'')$. The result is kept in S_0''.

iv. For each *delayable* transition or signal delivery transition t_i'' in the enabled list, S_0'' is intersected with $\mathsf{restrict\text{-}delayable}(S', g_i'')$. The result is kept in S_0''.

S'' is the last value of S_0''.

The state (q', S'') is marked as a successor of (q, S) by the transition t_i' in the simulation graph. If the simulation graph is not finite otherwise, closure with respect to the maximal constant c used in clock comparisons in the SDL specification (or in the MSC and GOAL observers) can be first applied to the zone S'' ($S'' := \mathsf{close}(S'', c)$).

8.2.3 MSC and GOAL specifications

The handling of MSC and GOAL observers in parallel with an SDL specification induces some modifications in the successor computation algorithm presented in the previous section. The computations related to observers take place between steps 2.a and 2.b.

Intuitively, based on the state (q', S') and on the events generated by the previous discrete step t_i', the simulator evaluates the fireable transitions of each observer. For each observer, there may be several fireable transitions $u_1, ..., u_k$, with the guards $h_1, ..., h_k$, but the parts of the simulation state (q', S') on which these are fireable (i.e. $h_i \cap S'$, $i = \overline{1,k}$) must be disjoint. This condition ensures that the behavior of the observer is deterministic for every explicit state $(q', \mathbf{v}) \in (q', S')$, which is essential in the definition of timed property automata.

The state (q', S') is partitioned in (at most) $k+1$ parts: $(q', h_1 \cap S')$,...,$(q', h_k \cap S')$ and $(q', S' \setminus \bigcup_{j=1}^{k} h_j)$, each part generating a different successor due to the observer transition. On each part $(q', h_j \cap S')$, the result of triggering the observer transition is computed, in a step similar to the step 2.a of the algorithm from the previous section. The step 2.b, computing the temporal successors (time-succ) is subsequently applied to the $k+1$ resulted states, and there will be $k+1$ successors of the initial state (q, S) in the simulation graph.

Moreover, if there are several observers executed in parallel with a specification, the state is partitioned for the first observer, each substate is partitioned for the second observer, and so on.

8.3 User-level features

The simulation tool provides essentially the same interface as the commercial version of the Simulator [TEL00a]. The tool may be used in the two modes presented in §8.1: simulation (user-controlled or random) and verification (with finite or Büchi acceptance conditions).

Interactive commands and presentation of results

In simulation mode, the user guides interactively the execution of the model. A number of additional commands related to the language extensions are available for:

– *visualizing the clock zone* of the current simulation state. The command `clocks` prints the clock constraints defining the zone associated to the current simulation state. For example, the output of this command during the simulation of the SpaceWire system presented in §6.2.4 may look like this:

```
> clocks
0 <= iface1!SM!rc <= 10440
0 <= iface2!SM!rc <= 7780
0 <= iface1!SM!rc - iface2!SM!rc <= 2660
```

– *measuring the time span between two events* in interactive simulation. It is difficult (or sometimes impossible) to derive information about the time span between two simulation events by examining the clock zones in the two states. For this reason, the simulator allows the user to create and destroy *chronometers*, which are used for measuring time in such situations. A chronometer behaves like an SDL *clock*, except that it is introduced from the simulation console. By consulting the clock zone after several simulation steps, the constraints on the chronometer indicate the lower and upper limits of the elapsed time interval.

  A typical scenario for taking measurements is shown below:

```
> addclock chron        -- add chronometer, start interactive measuring
added chronometer chron from console

...                     -- simulation steps

> clocks chron          -- consult chronometer
15360 <= chron <= 23260
> delclock chron        -- remove chronometer
deleted chronometer chron from console
```

– *visualizing the contents of delaying channels queues.* The command `dchannels` prints information about the signals in transit through a channel, as shown in the example below:

```
> dchannels
contents of channel physical_link direction towards iface2 =
  1 =
    sender = iface1!TX(1)
    name = NChar
    NChar =
      p1 = datachar
```

– *controlling the behavior of some extensions*, such as lossy channels.

Verification features

In verification mode, the simulation graph is built (entirely or partially) without the intervention of the user, in a pre-established order of exploration (depth-first or breadth-first). The following types of properties are checked on the fly:

- *Absence of deadlocks.* The conditions in which deadlocks are signaled have been discussed in Step 1 of the successors computation algorithm, on page 150.
- *Invariance properties.* These properties are given by propositional logic formulas that must hold in each state of the system. A formula is formed of atoms α which can test:
  1. The *discrete state* of the SDL system, such as the value of a variable, the length of a queue, etc. In this case, the satisfaction of α by (q, S) is decided based on the discrete part q.
  2. The *values of clocks*, using the same form of atomic clock comparisons as in SDL transition guards. The satisfaction of such conditions α by (q, S) is verified by checking the inclusion of the zone S in the polyhedron represented by the condition α.
- *Linear properties* specified in MSC or GOAL. The verification methods for MSC and GOAL properties have been discussed in Chapter 7.

Part III

Applications, Conclusions and Perspectives

Chapter 9

Case studies

We have performed several case studies for validating the concepts and tools developed in the context of this work. Some of the examples that have been considered are benchmark examples, frequently used in the timed verification literature, such as the train-gate-controller system (described in [Alu91] and [Tri98]) or the Bounded Retransmission Protocol (BRP has been used as a benchmark example for several verification tools [GvdP96, HS96, Mat96]; time-specific aspects of the protocol have been studied in [DKRT97], using a timed automata based formalism and the Uppaal tool [LPY97, BLL+98]). These case studies have given good results concerning the expressivity of the formalisms (SDL for describing the models, and GOAL and MSC for describing properties) and the power of the analysis methods that are used.

Several case studies using real-life system specifications have also been considered. The reliable multicast transport protocol RMTP-II [PMR+00, WPT99] is currently being modeled at France Telecom R&D in the context of the INTERVAL project[1], using a set of SDL extensions similar to those proposed in this thesis, for evaluation purposes. Besides a good expressiveness of the proposed SDL extensions, the study has also pointed out some shortcomings of the extensions, such as the impossibility to use time measurements in auto-adaptive systems. Partial results obtained in another study, concerning a multimedia synchronization protocol by Ericsson, confirm these conclusions.

In the following, we will describe in more detail a study based on the SpaceWire protocol [sWG00]. In §9.1 we give an informal description of the protocol, taken from the protocol standard draft. In §9.2 we outline the problems encountered when specifying the protocol in SDL, and show how the extensions proposed in this thesis help in building a more precise specification. §9.3 shows how timed functional properties of the protocol are described and verified using GOAL and MSC. Finally, in §9.4 we draw conclusions from the mentioned case studies.

9.1 The SpaceWire protocol

SpaceWire [sWG00] is a protocol stack used by the European Space Agency to handle payload data on-board a spacecraft. The purpose of SpaceWire is to provide a unified high-speed infrastructure for connecting together sensors, processing elements, mass memory units and other sub-systems. The standard covers several protocol layers, from the physical link up to the net-

[1] European project INTERVAL (IST-1999-11557): *Formal Design, Validation and Testing of Real-Time Telecommunications Systems.* http://www.cordis.lu/ist/projects/99-11557.htm

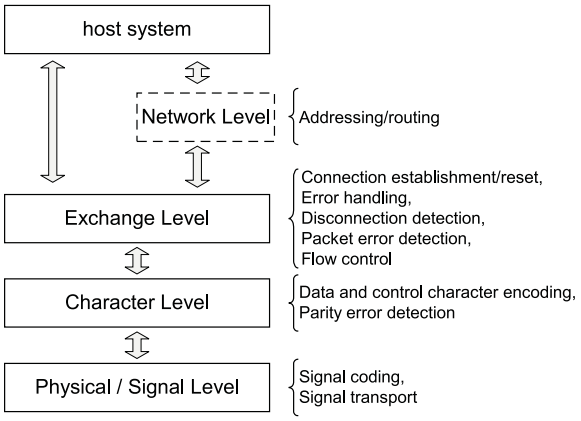


Figure 9.1: SpaceWire protocol layers and their functionality

work level. In our case study, we have focused on the exchange level, which corresponds to the data link level in the OSI stack. Parts of this specification were used throughout the previous chapters of this document.

In Fig. 9.1 we outline the layers of the SpaceWire protocol, and the functions fulfilled by each layer. The Exchange Level is implemented by a *link interface*, which makes the connection between a host system (directly or through the SpaceWire Network Level) and a physical link (through an additional bit encoding layer – the Character Level). Two link interfaces exchange full characters (as represented by the Character Level) over an *unreliable full-duplex point-to-point* link.

The functionality provided by an Exchange Level link interface is:

1. *Connection establishment.* Upon initialization, error or a soft reset (received from the host system), a link interface executes a reset cycle which is described further in this section. The reset cycle is partly time-controlled, and its purpose is to synchronize the link interfaces at the two ends, and to bring both of them back in the connected state.

2. *Error detection* through parity checking. Parity checking is actually handled by the underlying layer (Character Level), the Exchange Level only re-initializes the connection upon a parity error. No re-transmission functionality is provided.

3. *Disconnection detection* by continuous transmission of control characters on the transmitting side, and timeout detection on the receiving side.

4. *Flow control.* In order to avoid buffer overflows, each link interface will keep a credit counter, which represents the maximum number of data characters it is allowed to send to the other side. The counter is increased when the other side signals (using a control character) that new places are available in its receiving buffer, and decremented whenever a data character is sent over.

As defined by the standard, a link interface has three components: a transmitter (TX), a receiver (RX) and a state machine (SM). In the following, we show how this functionality described above is achieved by the components link interface.

The transmitter (TX)

The transmitter is responsible for sending data and control characters over the link, according to the data received from the host system and to the instructions from the SM. There are two kinds of control characters used by the Exchange Level:

- the NULL character, which is sent continually by the TX as long as the connection is established and there are no other characters to be sent. The NULL character is also used for synchronization in the initial phases of the connection.
- the FCT character, which is used for flow control. The FCT is used in order to signal that space for 8 more characters is available in the host receiving buffer. FCTs are sent by the TX at the request of the host system.

The TX also handles flow control information: each time the RX of an interface receives a FCT from the remote side, the FCT is forwarded to the TX, which increments a credit counter by 8. Each time the TX transmits a data character, the credit counter is decremented by 1, and data characters are sent only as long as the credit counter is positive.

The TX has 4 modes of operation:

1. *disabled* – no character of any kind is transmitted.
2. *sending NULLs* – in which only NULL characters are transmitted (continually) over the link.
3. *sending NULLs and FCTs* – in which NULL characters are transmitted (continually) over the link, and FCTs are transmitted at the request of the host system.
4. *sending NULLs, FCTs and NChars* – in which NULL characters are transmitted (continually) over the link, and FCTs and normal data characters (NChars) are transmitted at the request of the host system.

The TX switches between these modes of operation, at the request of the SM. They are used in the link interface initialization cycle, described further on.

The receiver (RX)

The function of the receiver is to forward the characters received from the link to either the SM, the TX, or the host system. Normally characters are handled as follows:

- NULLs are ignored, except for the first NULL received after a link re-initialization (which is forwarded to the SM),
- FCTs are forwarded to the TX and to the SM
- Normal characters are forwarded to the host system. However, the receiver takes care so that two end-of-packet characters are not received one after the other. If this is the case, an *empty packet error* is signaled to the host and to the SM (the latter will consequently re-initialize the link).
- Characters containing parity errors are signaled to the SM which will re-initialize the link.

The RX also uses a disconnection timer, to detect link problems. Whenever a period of 850ns elapses without a character being received, this is signaled to the SM which will re-initialize the link. The disconnection detection mechanism is enabled when the first character is received.

The RX has only two functioning modes:

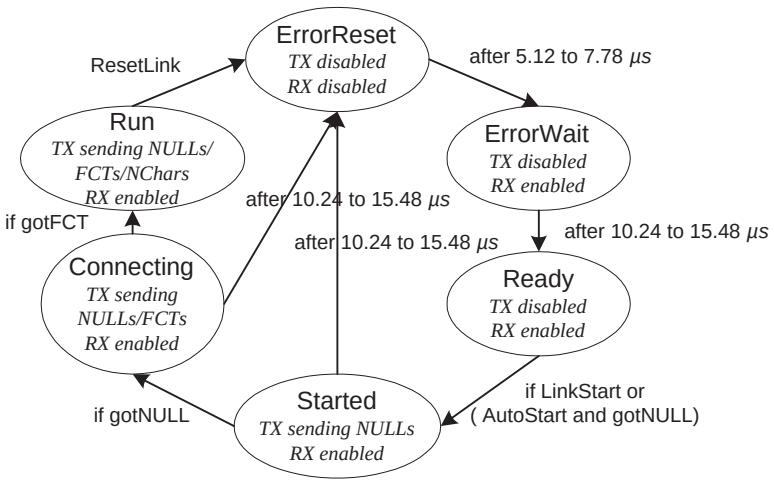


Figure 9.2: SpaceWire link interface initialization cycle

1. *disabled* – all characters are ignored
2. *enabled* – characters are handled as shown above. Additionally, the RX ignores all the characters received before the first NULL character after a link initialization.

The state machine (SM)

The main function of the state machine is to provide the synchronization logic for link establishment. When a link is established, the SM's of the interfaces at the two ends remain in the same state without taking any action (until an error occurs or the respective interface is reset by the host system).

The reset cycle executed by a link interface (at initialization time, or after an error or a soft reset) is depicted in Fig. 9.2. The phases (states) of the cycle are:

1. **ErrorReset** – This state is entered at initialization time, after an error or after soft reset. In this state, the SM will disable both the TX and the RX. This state is normally left after a $6.4\mu s$, but the standard allows a jitter, so the actual value may be between $5.12\mu s$ and $7.78\mu s$.

   During this period, if the remote interface was still connected it will detect a disconnection (as no NULL or other character is transmitted for more than 850ns), and will initiate the reset cycle.

2. **ErrorWait** – In this state, the RX is enabled, and it begins listening for NULL characters. If a NULL character is received, the gotNULL condition (used in state **Started**) is set to true. The state is left after a nominal delay of $12.8\mu s$ (that is between $10.24\mu s$ and $15.48\mu s$ with the jitter). The delays in **ErrorReset** and **ErrorWait** are chosen so that receivers at both ends are enabled before either end begins transmission.

3. **Ready** – This is a transient state which is left as soon as the link may be initialized. A link interface may run in two modes: *LinkStart* – in which the interface does not have to wait for an external event in order to be initialized, and *AutoStart* – which is a slave

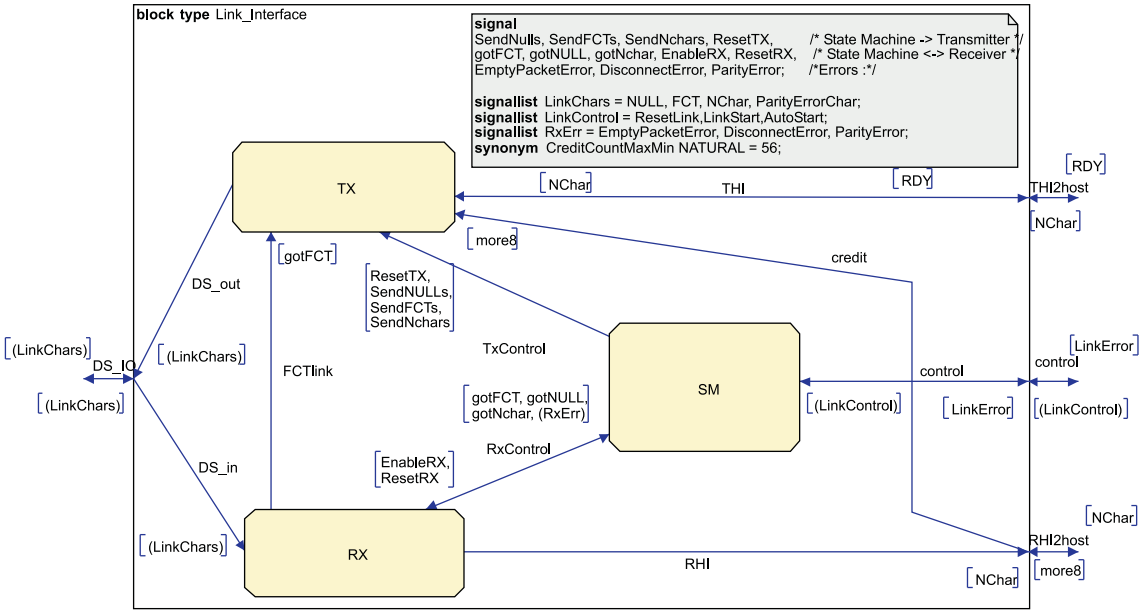


Figure 9.3: SDL modeling of a SpaceWire link interface

mode, in which the link is initialized when the first NULL is received. The condition for passing from **Ready** to **Started** is therefore: (*LinkStart* **or** (*AutoStart* **and** *gotNULL*)).

4. **Started** – In this state, the TX switched to *sending NULLs*. If gotNULL is already true, or as soon as the first NULL is received, the SM is switched to state **Connecting**. If no NULL is received within 10.24 to 15.48μs , the interface is reset again.
5. **Connecting** – In this state, the transmission of FCTs is allowed, and the interface waits for a FCT from the remote side. The FCT means that the remote side is ready to receive data characters, and therefore the interface may be switched to **Run**. If this condition is not met within 10.24 to 15.48μs, the link is reset again.
6. **Run** – This is the normal operation state, in which all control and data characters may be sent. This state is left only if the host resets the link, or if an error occurs.

Additionally to the transitions represented in Fig. 9.2, transitions to the **ErrorReset** state may be taken in any state upon the receipt of a character containing a parity error, or the receipt of an unexpected character (NChar in a state different from **Run**, FCT in a state different from **Run** or **Connecting**).

9.2 SDL modeling and expressivity problems

Although the Exchange Level of the SpaceWire protocol is designed to be implementable in hardware, the signaling between the components of a SpaceWire interface is asynchronous, and SDL is a suitable choice for modeling and validating the functional and timing aspects of the protocol. In the following we discuss the SDL modeling of a SpaceWire link interface, the validation model built around it, and we outline some of the problems occurring if the standard version of the language is used for modeling.

The link interface (Fig. 9.3) is modeled as a block agent, with three component processes corresponding respectively to the transmitter (TX), the receiver (RX) and the state machine (SM). On one side, the link interface is connected to the physical link through the bit encoding level, which is abstracted away our modeling. Thus, two link interfaces connected together exchange full characters, represented by the SDL signals *NULL*, *FCT*, *NChar*, and *ParityErrorChar*. *ParityErrorChar* is introduced to represent characters received with a parity error. Moreover, as the functioning of the protocol does not depend on the content of data characters (*NChar*), the only information carried by *NChar* signals is whether the character is a normal data character or an end-of-packet (EOP,EEP) character.

On the other side, the link interface communicates with the host system through several gates:

- *RHI2host* – provides a receiving interface by which *NChars* are received, and credit signals (*more8*) are sent.
- *TIH2host* – provides a transmit interface by which *NChars* are sent, and by which the interface signals to the host system when it is ready to transmit (RDY).
- *control* – provides an interface for link control, transferring signals for: resetting the link, establishing the link operation mode, signaling link errors.

The specification of TX, RX and SM are shown in Fig. 9.4-9.6. Their behavior corresponds in a straightforward manner to that described by the standard:

- The *TX* has four states corresponding to the four operating modes described in the protocol standard. An additional state (*stopped0*) is necessary in order to flush the signal queue when the *TX* is reset. This action is currently not prescribed by the standard, but during the verification we discovered that the protocol functions incorrectly if signals are preserved in the queue after a reset.

  The manipulation of the credit counters used for flow control is specified on the transitions from state *TX*.

  Communication delays, which are important for the timing of the protocol, are specified by the standard and depend on speed of the link, which may be 100, 200 or 400 Mbps. In the SDL specification we have considered the case of a link operating at 100 Mbps, and we have taken the nanosecond as time unit. The communication delays could not be modeled using the extensions for delaying channels proposed in Chapter 6 because, on one hand, characters have different lengths (a *NChar* has 10 bits, a *NULL* has 8 bits, and an FCT has 4 bits), and on the other hand the *TX* should be blocked while transmitting a character, which is not the case if a delaying channel is used. Communication errors are also modeled explicitly in the *TX*, as the assumption about the physical SpaceWire link is that single-bit errors may occur and are detected using a parity bit.
- The *RX* has a state (*NotEnabled*) corresponding to the *disabled* operating mode (see previous section), and two states (*WaitNull* and *Idle*) corresponding to the *enabled* mode. In *WaitNull*, the RX waits for the first *NULL* character received after a link reset. In normal operation mode (*Idle*), the *RX* handles all types of characters that may be received from the lower layer.

  The disconnection timeout is modeled using a clock (*dt*) and a *delayable* transition, because the standard allows a range of values between 740 and 1080*ns* for the timeout.

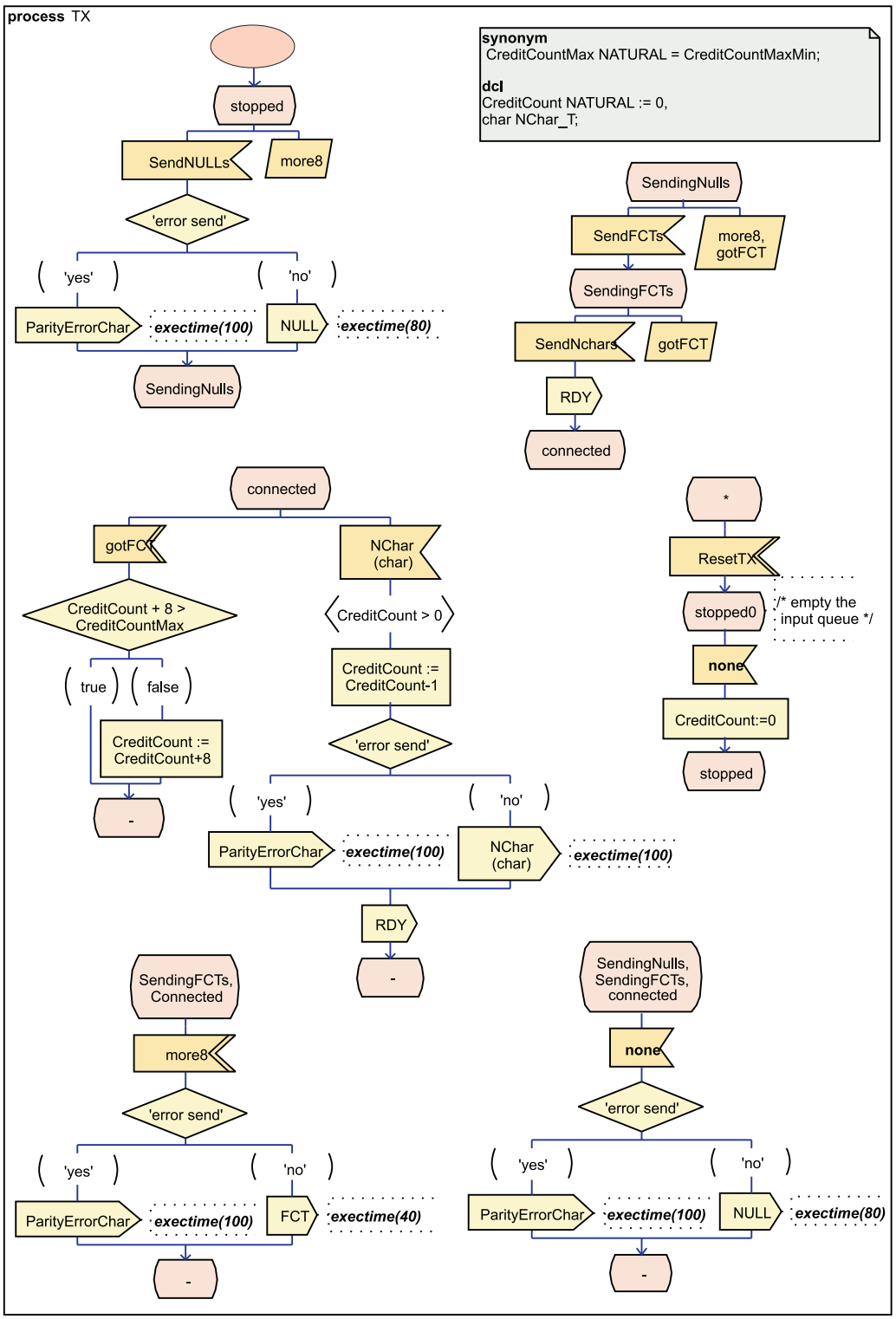


Figure 9.4: SDL model for the *TX*

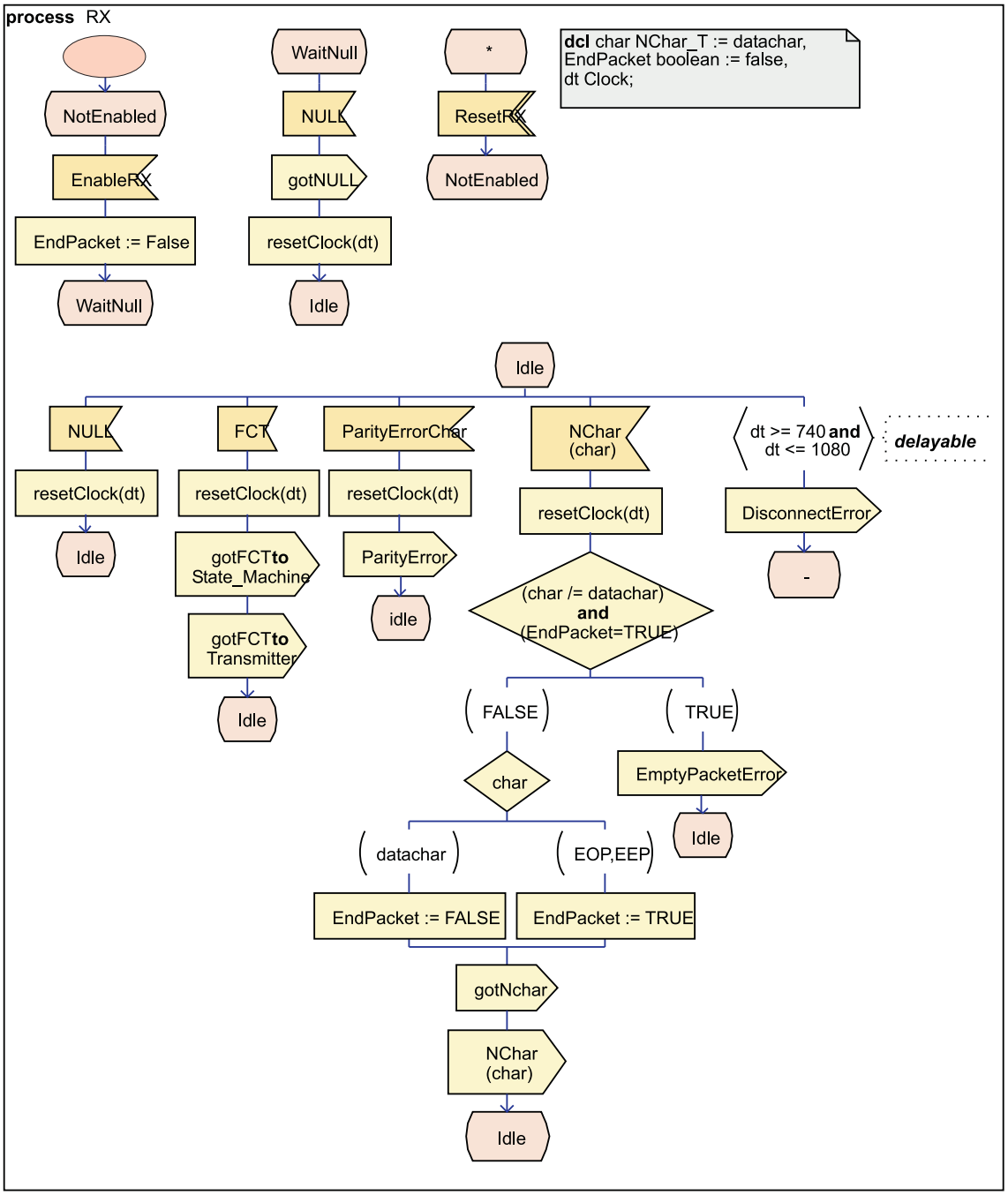


Figure 9.5: SDL model for the RX

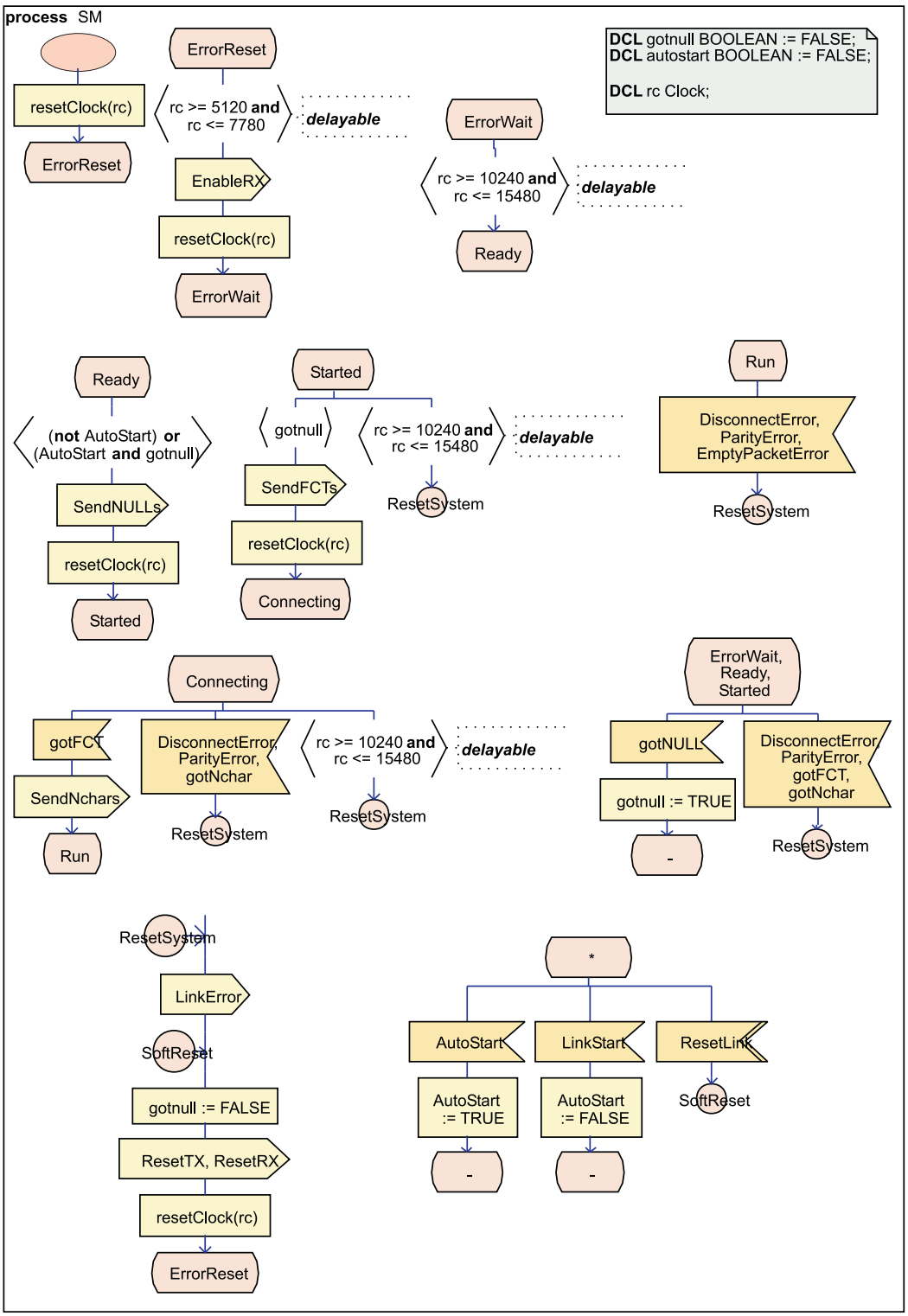


Figure 9.6: SDL model for the *SM*

- The *SM* process has 6 states corresponding to the operating modes described in the previous section. The transitions correspond directly to the transitions shown in Fig. 9.2.

  The time-triggered transitions specified in the standard are modeled using *delayable* transitions, in order to capture the allowed non-determinism.

The standard semantics of SDL, which makes no assumption about the processing times of agents, cannot be used for validating the SpaceWire specification. The reason is that the protocol relies on strict reaction times of each component, and has no mechanism for detecting malfunctions due to slow reactions of certain components. Using the standard semantics of SDL, nothing can be ensured concerning the functioning of the link (e.g. it cannot be ensured that, in the absence of errors, the link will eventually be established, as it could take an infinite time for the *RX* at one end to detect the first *NULL* character received).

The semantics of SDL provided by most simulation and verification tools is closer to the needs of this specification, since assuming 0-reaction time is reasonable for this specification. However, this semantics is not sufficient for validating the functioning of a link in all cases, because the SpaceWire standard allows large ranges for every timer or duration used in the specification. Using the standard constructs of SDL, only one combination of time values used in the SM's may be validated at a time.

9.3 Verification

We have used the SDL specification of SpaceWire presented in the previous section in two ways:

- First, we exploit the capability of GOAL observers to generate traces, in order to make end-to-end *time measurements* for specific functions of the protocol.
- In a second phase, we use the timing information acquired before to construct and verify combined *functional and timing properties* of the protocol.

The validation model

For validating the functioning of a link interface, we had to model the environment in which the link operates. The SDL model built for validation purposes includes two link interfaces connected through a physical link, as well as the two host systems controlling the interfaces. The structure of the validation model is shown in Fig. 9.7.

For the physical link, since bit errors and communication delays are modeled in the *TX*, we only need to model the possibility for the link to break up; this is characterized by the complete loss of signal for an undetermined amount of time, which we may model using the *lossy* channel construct.

For the hosts, we must assume the weakest hypotheses about their behavior, in order to obtain results that hold in most real case. The behavior of hosts in the verification model is shown in Fig. 9.8. At initialization, a host establishes the operating mode of the link (*AutoStart* or *LinkStart*). If *LinkStart* is chosen, then the host may initialize the link after an indeterminate amount of time; this is modeled using a *lazy* transition. After the initialization, the host sends within a bounded amount of time a credit signal (*more8*), then passes in a normal operation mode (represented by state *RecvTrans*).

In order to verify the flow control mechanism, the host too keeps a credit counter (*cpt*), which is increased when a *more8* is sent, and decreased when a *NChar* is received. If this

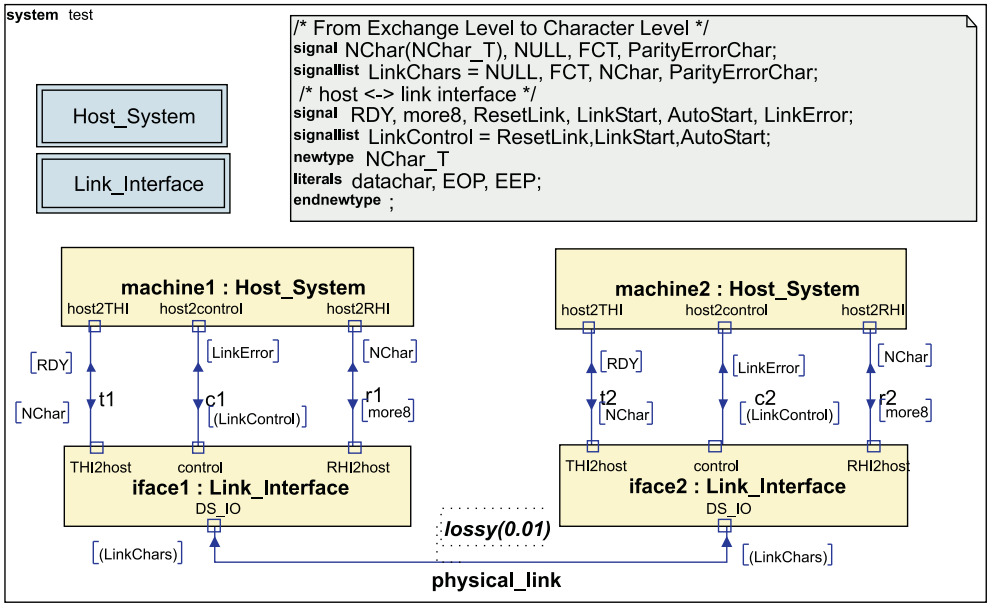


Figure 9.7: The SpaceWire validation model

counter reaches a value below 0, it means that a buffer overflow has occurred in the host. Normally, the protocol should ensure that this does not happen.

The host also performs several actions at randomly chosen moments:

- it sends data characters at random (after having received a *RDY* signal from the link interface),
- it sends *more8* signals at random. To limit the explored model, we allow this only when $cpt = 0$, so cpt is never greater than 8.
- it may reset the link at random.

These actions are modeled using *lazy* transitions, to allow them to happen at arbitrary moments.

Time measurements

The global correct operation of a SpaceWire link is a complex property, that depends on the fulfillment of several simpler properties concerning each of the four basic functionalities of the protocol enumerated on page 158. While the functional properties that the protocol must satisfy (related to each functionality) are easily deduced from the standard, there are no user-defined requirements concerning timing. For this reason, in our case study we started by making time measurements for the basic functions of the protocol, and we used the resulted values to construct correctness properties.

From the user's point of view, an interesting characteristic of a SpaceWire link is the minimal/maximal time it takes for the connection to be established after a reset or an error. This depends on the operation mode of the link interfaces at the two ends. In the following, we show how measurements were made in the case of two link interfaces, one operating in *AutoStart* mode and the other operating in *LinkStart* mode, in the absence of transmission errors.

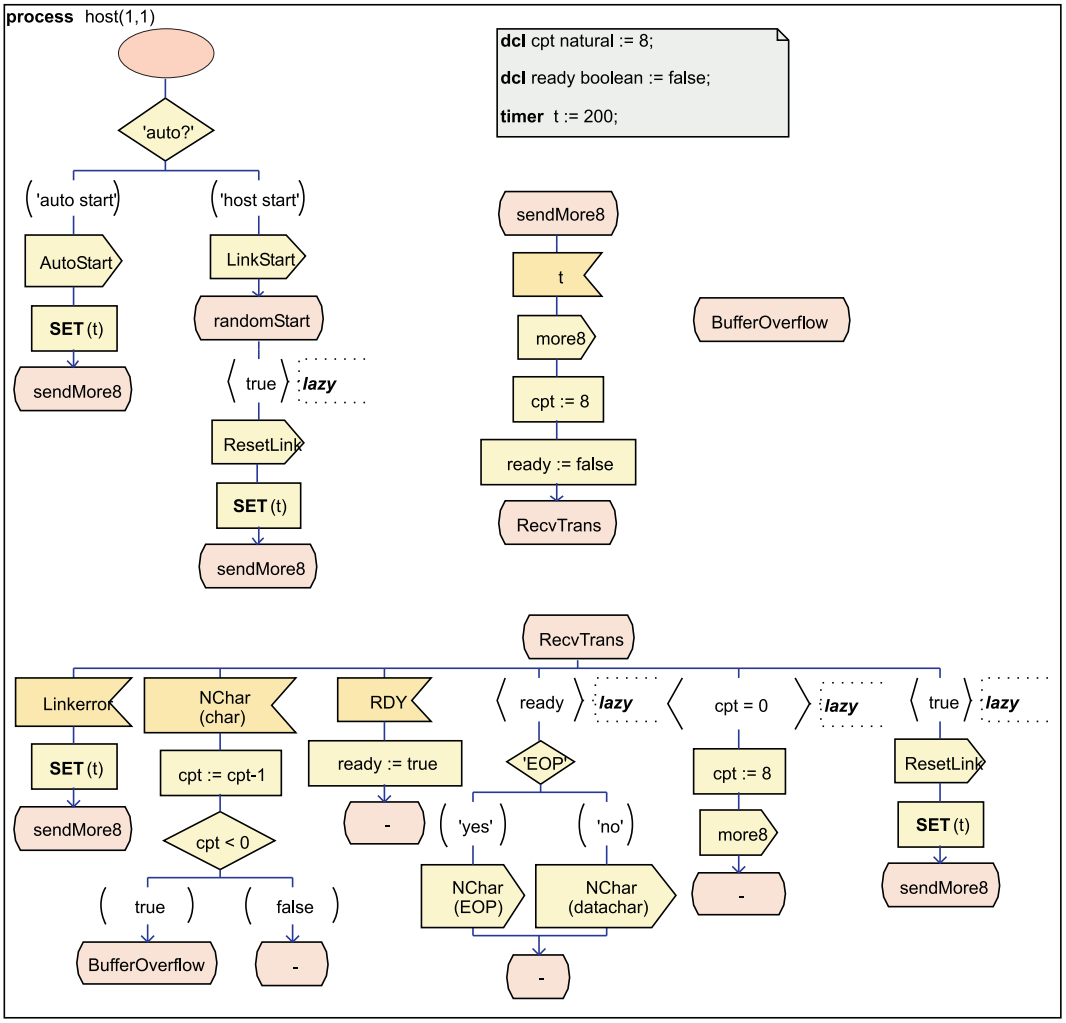


Figure 9.8: The specification of a host

Using the existing mechanisms of the *Object*GEODE simulation tool (notably, transition filters), we guide the exploration of the SDL model so that the above hypotheses are met. The observer in Fig. 9.9 is then used to measure the time between the sending of the *ResetLink* signal by the host functioning in *LinkStart* mode, and the moment when the state *Run* is reached by both interfaces (and therefore the connection is established). The `writeln(z)` statement in the GOAL observer outputs the minimal and maximal bounds on the clock z, as specified by the clock zone of the current state. By performing an exhaustive exploration of the state space, we will obtain several minimal and maximal bounds, corresponding to all scenarios through which the connection is established. The global minimal and maximal bounds may then be computed.

With the hypotheses assumed before, we obtain a minimal value of $15560ns$, and a maximal value of $24390ns$ for the connection establishment time. However, during this experiment we discovered that the host system has to satisfy a number of timing constraints to ensure the establishment of the connection in a bounded time:

– Each host should send a first credit signal (*more8*) within a bounded amount of time after

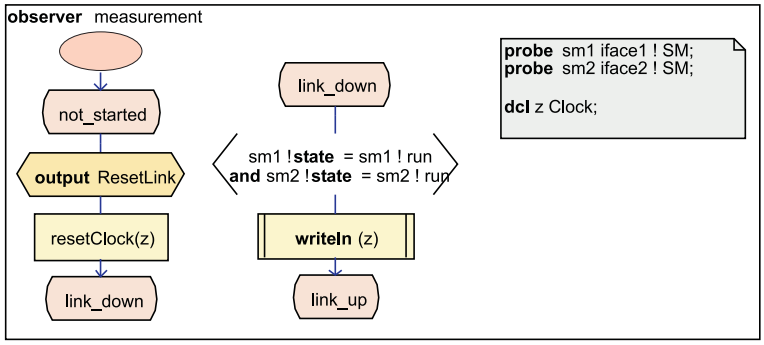


Figure 9.9: Observer for measuring connection time

the (re-)initialization of a link interface. Initially we modeled the sending of the first *more8* signal with a *lazy* transition. However, in that case a link interface may never sent a *FCT* signal over the link, causing the opposite interface to be unable to take the transition from the *Connecting* to the *Run* state.

– After a link reset, a host should let enough time for the *TX* to be reset, before sending the credit signal (more8). The reason is that, if the *TX* is busy sending a character when it receives a *RexetTX* signal, the reset will be taken into account only after the current character is transmitted. If in the meantime a *more8* is received, it will be lost during the reset process that follows. Therefore, the *more8* signal should be sent after at least the maximal character transmission time ($100ns$ on a 100Mbps link) from a reset.

For these reasons, a credit signal is sent $200ns$ after every reset or initialization, in our model. The minimal and maximal times obtained above hold with this hypothesis.

Other time measurements, for example concerning the connection times under different assumptions or the error detection times, may be made using the previously described method.

Verification of timing properties

We have verified several timed functional properties on the SDL specification of SpaceWire. We discuss below a property referring to the establishment of a SpaceWire connection. The exact form of the property expressing correct connection establishment depends on the configuration of the hosts and on the errors occurring on the physical link. We consider here the case when one host is configured in *LinkStart* mode, and the other in *AutoStart*, and no errors occur on the physical link.

In this case, when the host configured in *LinkStart* mode sends the *ResetLink* message, the connection should occur in at most 24390ns, as indicated by the measurements described in the previous section. This property, expressed in a variant of linear temporal logic with bounded time operators[2] would have the following form:

$$\varphi = \Box\,(\langle \text{output ResetLink} \rangle \rightarrow \Diamond_{\leq 24390} \langle \text{Link Up} \rangle)$$

In the above formula, $\langle$output ResetLink$\rangle$ represents an atomic proposition which holds in the current state if that follows after a transition in which the *ResetLink* signal was sent. $\langle$Link Up$\rangle$

[2]Such operators are used in the quantitative branching time logic TCTL [ACD93]. See also the survey [AH91]

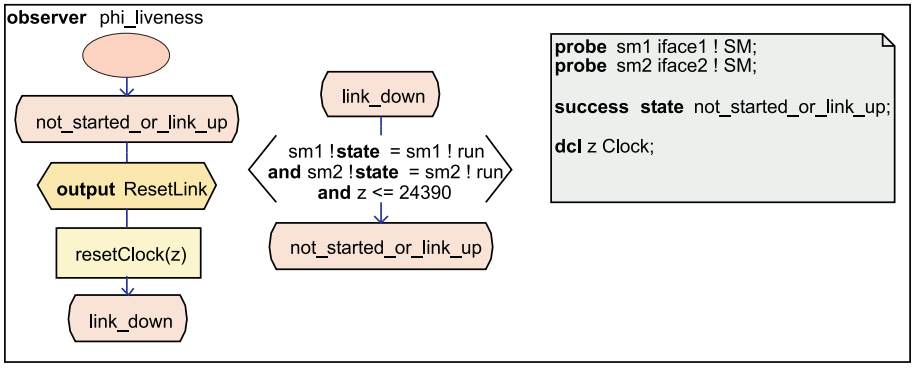


Figure 9.10: Observer for verifying connection establishment in a particular configuration

is an atomic proposition representing the states in which the connection is established. (Note that in the classical definition of LTL, atomic properties like ⟨output ResetLink⟩ cannot be represented, as they depend both on the state of the model and on the previous transition by which this state was entered. However, the definition of LTL formula satisfaction may be adapted to accommodate such atomic propositions.)

In GOAL, the ⟨output ResetLink⟩ atomic proposition is equivalent to the "**when output** ResetLink" observation predicate. The value of ⟨Link Up⟩ is given by the following condition on the components of the SDL model state:

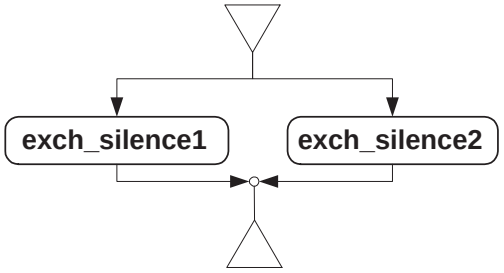
$$\langle \text{Link Up} \rangle \equiv (\text{sm1 ! } \mathbf{state} = \text{sm1 ! run } \mathbf{and} \text{ sm2 ! } \mathbf{state} = \text{sm2 ! run})$$

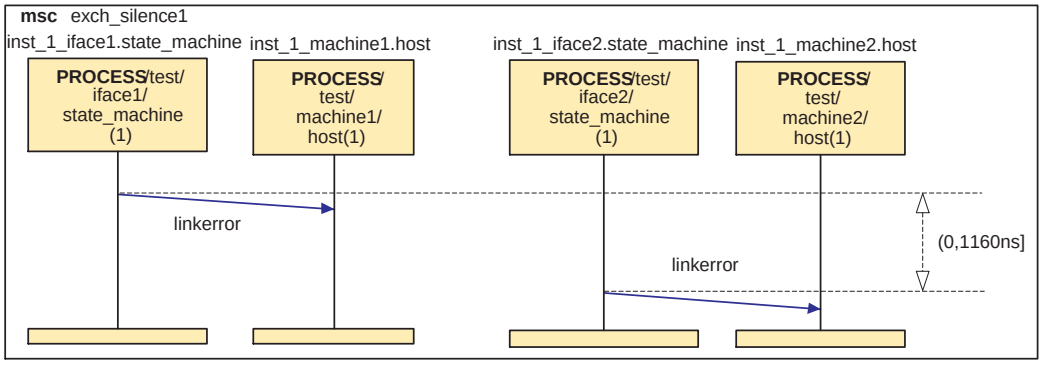
The property φ may be expressed as a GOAL liveness property, as shown in Fig. 9.10.
Other verified properties refer to different functions of the protocol, such as:

1. *Exchange of silence.* As shown in the specification, the re-initialization of both sides of a link after an error is based on the exchange of silence: the side detecting the error enters the reset cycle (by the *ErrorReset* state); the other side will not receive any more *NULL*s, and will eventually detect a disconnection timeout and enter the reset cycle.

   This property may be expressed more formally, for example, as follows: an exchange of the signal *LinkError* inside one link interface is eventually followed, within a bounded amount of time, by the exchange of a *LinkError* signal in the opposite link interface. The amount of time is more precisely $1160ns$, which is the maximal duration of the disconnection timer plus the transmission time of a *NULL* character which may begin just before the first *LinkError* signal. In the following, we show how this kind of properties, which involve only exchanged messages and associated timing information, may be expressed and verified using MSC specifications.

   Consider the HMSC with two alternatives represented in Fig. 9.11. The first alternative expresses the fact that after the exchange of a *LinkError* signal in *iface1*, another *LinkError* signal is exchanged in *iface2* within at most $1160ns$. The second alternative specifies the same events in the reversed order. Then, the timed property automaton corresponding to the HMSC *silence* may be used for verifying correct exchange of silence, using the method described in §7.2.3 (that is, provided that the initial state of the TPA corresponding to the HMSC is considered a *success* state, so that systems in which no error occurs are also considered correct).





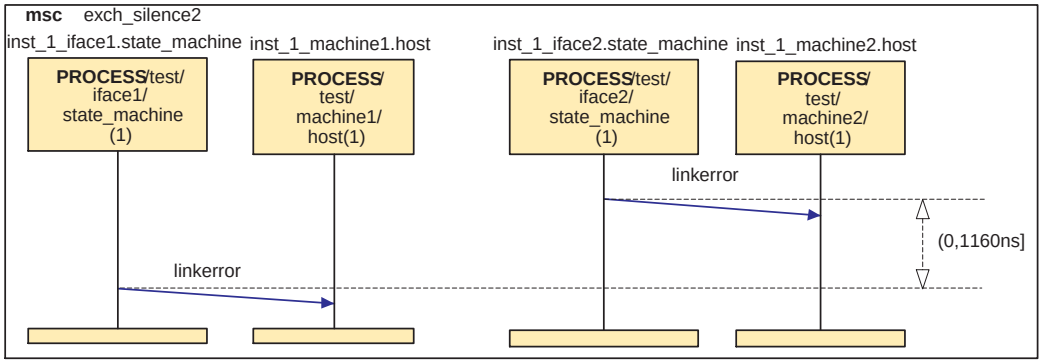


Figure 9.11: MSC property expressing correct exchange of silence

2. *Flow control.* The correct functioning of the flow control scheme of SpaceWire does not involve timing. The property that has to be satisfied is that the hosts never reach the *BufferOverflow* state. This is a simple safety property that may be verified using only invariants, supported by the *Object*GEODE verification tool.

   The property is satisfied if no error or only bit (parity) errors occur on the physical link. However, if the link is completely interrupted for a period less than that of the disconnection timer, after which the link comes back into operation, this error is not detected and does not produce a re-initialization. In this case, if a *FCT* character is lost while the link is down, the flow control scheme may no longer function correctly.

9.4 Conclusions

The case studies performed show that the language extensions introduced in this work are able to capture timing information both in the models and when writing temporal properties. They have also pointed out some of the limits of the model, related to:

1. the *abstraction level*: in some cases, the primitives are too low level, requiring a more complicated modeling. This is the case with the modeling of transmission delays in SpaceWire, or with the modeling of congestion (dynamically changing execution times) in other models.
2. the *expressivity* of the primitives: as pointed out by the RMTP-II case study mentioned in the beginning of this chapter, systems which adapt their behavior based on results of time measurements cannot be modeled using the primitives introduced in this document.

On the tool side, the case studies have pointed out the necessity for applying state space reduction techniques in case of large specifications. For example, we have been able to analyze the SpaceWire model only after (manually) performing a live variable analysis and reduction. Several other types of static analysis and reduction techniques are suggested in [Boz99] for a formalism similar to SDL (IF). They provide very good reduction ratios and could be easily adapted for SDL.

As such techniques are not yet integrated in our tool, we considered it premature to make performance comparisons between the extended *Object*GEODE verification tool and other timed model analysis tools.

Chapter 10

Conclusions and perspectives

The work presented in this thesis deals with the integration of timed modeling and validation techniques within the framework of SDL. The concrete results of this work are situated at several levels.

At the system specification level, we have defined a set of extensions which enable the modeling of timing assumptions and of non-trivial time-dependent behavior in SDL. We have also proposed an alternative semantics of time in SDL, which allows model-based tools such as simulators and verifiers to use more realistic assumptions about time progress, when analyzing a system specification. These assumptions are derived from the timing annotations introduced in the specification. We note that the standard semantics of SDL, which constitutes the starting point of the semantics described in this thesis, uses very loose assumptions about time progress and therefore cannot be used for validating quantitative timing properties.

At the property specification level, we have studied two languages commonly used for writing properties of SDL models, GOAL and MSC. GOAL is an automata-like observer language used in the *Object*GEODE tool [TEL00a] for specifying properties of SDL systems, as well as for controlling the simulation and verification process. In this thesis we have proposed a concise set of language extensions which enable the specification of quantitative timing properties in GOAL. We have also studied its semantics, and the satisfaction relationship between SDL models and GOAL properties. In order to provide a sound semantic basis for these, we have defined an abstract model of properties, the timed property automata (TPA), and its relation to timed automata (which form the semantic basis of SDL in our framework).

On the side of MSC, we have been confronted with problems of a different nature. MSC is a standard language for representing system execution traces. MSC is usually employed for modeling requirements, representing selected traces, etc., as an informative counterpart to a system specification. Although very expressive, the language cannot be used as a formal property specification language, for two reasons: it does not have a formal semantics in its latest revision (MSC-2000) which includes constructs for modeling timing constraints, and it lacks an interpretation as a property language (that is, a clearly defined satisfaction relationship between system models and MSC specifications). The results of our study cover both directions mentioned before. We propose a *semantics* for a (regular) subset of MSC-2000 which includes timing aspects. This semantics is based on timed automata, and is inspired by the (non-timed) Petri Net semantics of MSC proposed in [GPR93]. We presented in this document only the main lines of the semantics, without completely formalizing the definition. Concerning the satisfaction relationships between system models and MSC specifications, we discuss several alternative definitions for them.

At the level of analysis methods and algorithms, we have first studied the model checking problem for timed property automata. The obtained results show what abstractions may be used for deciding TPA satisfaction, and how classic Büchi automata-based model checking algorithms may be adapted for verifying TPA properties. These results, projected at the level of GOAL and MSC, provide model checking methods for these two languages.

An important result of this work concerns the abstraction used for TPA model checking. This abstraction, a variant of the timed automata *simulation graph*, is more complicated than the original version presented in [Tri98], as it has to accommodate several new constructs allowed in SDL models. We provide both an algorithm for building the extended simulation graph, and a correctness proof for the formulas used in the algorithm.

In order to assess the power of the language extensions and of the analysis techniques proposed in this work, we have implemented them in a tool prototype derived from an industrial SDL simulation and verification tool. The advantage of starting from a full-fledged tool is that the implementation of constructs existing in the standard SDL language may be reused, which diminishes the number of limitations of the tool and eases experimentation. A number of new features implemented in the tool have been inspired by the case studies on which the tool was applied.

The case studies have shown that the SDL language extensions are capable of capturing many forms of timing constraints appearing in real-time system specifications. Additionally, GOAL and MSC prove to be very intuitive formalisms for describing quantitative timing properties. While GOAL is very flexible, however, the lack of a standard interpretation of MSC as property language is a shortcoming.

The application of the tool on case studies confirms that the proposed analysis methods allow the derivation of interesting timing information, such as minimal/maximal delays between different events occurring in a system. An example is the SpaceWire study presented in Chapter 9. This case study shows that an accurate modeling of the system timing may provide additional insight in the functioning of the system, and may for example reveal hidden timing dependencies between system components.

The case studies have also pointed out some limitations of the proposed techniques. The language extensions made here prove to be low level in some cases, and unable to express certain forms of timing constraints, as explained in Chapter 9. The analysis techniques also are sometimes expensive in terms of computation, especially in the case of systems in which the analysis yields non-convex clock polyhedra.

Perspectives

There are several directions on which this work may be continued, for achieving a wider integration of timing modeling and validation methods in industrial development frameworks.

Higher level modeling constructs. When experimenting with the SDL language extensions proposed in this thesis, we realized that they are sometimes ill adapted and counterintuitive for a user who is not accustomed with the underlying semantic concepts. For example, the *urgency* concept provides a flexible way of specifying the moment when an event (e.g. a transition) occurs in a system, but the rules for deriving the urgency information from the high-level system requirements are difficult to formalize and may be misleading for the SDL modeler. On the other hand, some extensions introduced here are too basic and not flexible enough. It is the case for

channel specifications, which cannot model for example delays depending on the type of the signal, nor other types of transmission errors except signal loss.

For this reason, a promising work direction is the definition of a set of higher-level constructs for modeling timing information, to be included in SDL, based on the same semantic concepts as the primitives described in this work. These should be closer to the abstraction level of SDL, and should reflect more directly the kinds of (timing) information usually appearing in real-time system requirements.

Improvement of verification techniques. As noted in the conclusion of Chapter 9, there is a practical necessity to apply state space reduction techniques in connection with the verification method proposed in this work. A reduction method that may be easily adapted in this framework is the elimination of inactive variables. This method is likely to produce good results in the case of SDL, as the language defines several implicit components of the model state (e.g. the **sender** implicit variable attached to each SDL agent) which contribute to the explosion of the state space even when they are not used in a model.

Other reduction methods include partial orders, which attempt to avoid the exploration of unnecessary interleavings of independent transitions. Such methods have been successfully applied in the analysis of non-timed systems. However, their results in the case of timed models are less spectacular. A reason for this is that the time progress condition in a timed model, like the SDL model proposed in this work, is a global condition depending on the state of all system components, which reduces the overall independence between transitions of different components. Nevertheless, recent research results [BJLY98, Pag96, Min99] set hopes for a successful application of partial order techniques on timed models.

Application of new validation methods. A direction in which we plan to pursuit this research is the application of other validation techniques on timed models. We are thinking primarily at testing, which is at the moment the most used validation method in industrial development.

A research field that has been continually developing over the past decade is automated test generation from formal system specifications. The *Object*GEODE framework, on which the tool presented in this thesis is based, includes a test generation tool (TestComposer, [KJG99, KO99]) which is based on the exploration of the state space of an SDL model, using a technique similar to that of a verification tool. Currently, the tests generated by the tool contain information only on the discrete events exchanged between system components and the environment. We are thinking about extending the tool to include selected timing information in the generated tests. The main problem is to provide a flexible mechanism through which the user might select the timing information that is actually important for testing, from the quantity of information that a timed state space exploration provides.

On the side of test specification and execution, the available techniques are better prepared for tackling the testing of timed systems. In our previous work [OK99] we studied the problem of specifying timing information in a visual MSC-based test specification formalism, and took an approach similar to that of the standard test description language TTCN [ISO92]. However, we are aware of ongoing research aiming to improve the support for testing real-time requirements in TTCN [WG97, HKN01].

Integration within other languages. The spectrum of languages preponderantly used in the industrial production of real-time systems is continually changing, and newer languages

are becoming de facto standards. This is the case of the Unified Modeling Language (UML, [OMG99]), which has a growing importance in all branches of system modeling. Hence, the possibility to adapt the modeling and analysis techniques presented in this work to new languages is an important issue.

A preliminary study [Obe00] we have performed concerning the application of these methods to UML shows no apparent incompatibility. UML uses a state machine based approach for specifying the behavior of system components, which can accommodate the constructs for specifying timing information introduced here. Moreover, the proposed UML profile for schedulability, performance and time [tadwg00] introduces some of the necessary modeling concepts, such as time values (that may be attached to certain constructs in the model), time events, etc., without however giving a clear semantics for these.

We also note that, because UML has several types of diagrams presenting different views of a system, it is more difficult to make a formal analysis of a system specification. This assertion is supported by the fact that all approaches to define a formal semantics for UML of which we are aware (see the overview in [Sta01]) tackle only a part of the language. Also, the language definition still contains a number of deficiencies. In this context, we have worked on the clarification of the concurrency model of UML [OS99], but other points need to be worked out in order to obtain specifications amenable to formal analysis. Among them, we mention: the semantics of actions, the semantics of time and the level of atomicity of transitions and actions, the construction of the initial system state, the exact semantics of communication between objects.

Nevertheless, as many efforts in the industry, academia and standardization bodies are put into making UML more formal and precise, the perspective of doing model-based timing analysis on UML models should become realistic in the long run.

Bibliography

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model checking in dense real time. *Information and Computation*, (104):2–34, 1993.

[ACH93] Rajeev Alur, Costas Courcoubetis, and T. Henzinger. Computing accumulated delays in real-time systems. In *Proceedings of the Fifth Conference on Computer-Aided Verification*, number 693 in LNCS. Springer-Verlag, 1993.

[ACH+95] Rajeev Alur, Costas Coucoubetis, Nicolas Halbwachs, Thomas A. Henzinger, P.H. Ho, X. Nicolin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, (138):3–34, 1995.

[ACHH93] Rajeev Alur, Costas Courcoubetis, T. A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems. In *Hybrid Systems II*, volume 736 of *LNCS*. Springer Verlag, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.

[AH91] Rajeev Alur and Thomas A. Henzinger. Logics and models of real-time: A survey. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer Verlag, June 1991.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 35–48. Springer Verlag, 1996.

[ALH95] B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL behaviors with GEODE. In R. Braek and A. Sarma, editors, *SDL'95 with MSC in CASE*. Elsevier Science B.V., 1995.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real Time Systems*. PhD thesis, Department of Computer Science, Stanford University, 1991.

[Ame87] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[AMP98] Eugene Asarin, Oded Maler, and Amir Pnueli. On the discretisation of delays in timed automata and digital circuits. In *Proceedings of CONCUR'98*, 1998.

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[Bö60] J. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[BAL97] H. Ben-Abdallah and S. Leue. Expressing and analyzing timing constraints in message sequence chart specifications. Technical Report 97-04, University of Waterloo, 1997.

[BAMP83] Moti Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BER94] Ahmed Bouajjani, R. Echahed, and R. Robbana. Verifying invariance properties of timed systems with duration variables. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*. Springer Verlag, 1994.

[BFG+99] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and Joseph Sifakis. IF: An intermediate representation for SDL and its applications. In R. Dssouli, G.v. Bochmann, and Y. Lahav, editors, *SDL '99. The Next Milenium. Proceedings of the 9th SDL Forum*, Montreal, Canada, 1999. Elsevier.

[BGK+00] Marius Bozga, Susanne Graf, Alain Kerbrat, Laurent Mounier, Iulian Ober, and Daniel Vincent. SDL for real-time: What is missing? In *The 2nd Workshop on SDL and MSC*, Grenoble, France, 2000.

[BGM+01] Marius Bozga, Susanne Graf, Laurent Mounier, Iulian Ober, Jean-Luc Roux, and Daniel Vincent. Timed extensions for SDL. In *Proceedings of the 10th SDL Forum*, LNCS, Copenhagen, 2001. Springer Verlag.

[BGMS98] M. Bozga, S. Graf, L. Mounier, and Joseph Sifakis. The intermediate representation IF. Technical report, Verimag, 1998.

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL With Applications from Protocol Specification*. The BCS practitioner series. Prentice Hall, 1991.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *Proceedings of CONCUR'98*, volume 1466 of *LNCS*. Springer Verlag, 1998.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Y. UPPAAL in 1995. In T. Margaria and B. Steffen, editors, *Proceedings of Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, Passau, Germany, March 1996. Springer-Verlag.

[BLL+98] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, W. Yi, and Carsten Weise. New generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, Aalborg, Denmark, July 1998.

[BMU98] J.A. Bergstra, C.A. Middelburg, and Y.S. Usenko. Discrete time process algebra and the semantics of SDL. Technical Report SEN-R9809, CWI, Amsterdam, 1998.

[Bor98] Sébastien Bornot. *De la composition de systèmes temporisés.* PhD thesis, Université Joseph Fourier, Grenoble, France, 1998. In French.

[Boy01] Marc Boyer. *Contribution à la modélisation des systèmes à temps contraint et application au multimédia.* PhD thesis, Université Toulouse III, 2001.

[Boz99] Marius Bozga. *Vérification Symbolique pour les Protocoles de Communication.* PhD thesis, Univ. Joseph Fourier, Grenoble, December 1999. In French.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, October 1998.

[Bro91] Manfred Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, (3), 1991.

[BS97] Sébastien Bornot and Joseph Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop HART'97*, volume 1201 of *LNCS*. Springer-Verlag, 1997.

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *Compositionality – the significant difference*, volume 1536 of *LNCS*. Springer Verlag, 1998.

[BW90] J.C.M. Baeten and W.P. Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, (18), 1990.

[BW95] Alan Burns and Andy Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier Science, 1995.

[CD94] Steve Cook and John D. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Object-Oriented Series. Prentice Hall, 1994.

[Cer92] Karlis Cerans. *Algorithmic Problems in Analysis of Real Time System Specifications.* PhD thesis, University of Latvia, Riga, Latvia, 1992.

[CES86] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[CHR92] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[CR96] Denis Caromel and Yves Roudier. Reactive programming in Eiffel//. In Jean-Pierre Briot, Jean-Marc Geib, and Akinori Yonezawa, editors, *Proccedings of OBPDC'95. Object-Based Parallel and Distributed Computation*, volume 1107 of *LNCS*, pages 125–147. Springer-Verlag, 1996.

[CVWY92] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Methods in System Design*, 1:275–288, 1992.

[DH98] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. Technical Report CS98-09, The Weizmann Institute of Science, Rehovot, Israel, April 1998.

[DHHMC95a] Marc Diefenbruch, E. Heck, Jörg Hintelmann, and Bruno Müller-Clostermann. Performance evaluation of SDL systems adjunct by queueing models. In R. Braek and A. Sarma, editors, *Proceedings of SDL Forum'95*. Elsevier Science B.V., 1995.

[DHHMC95b] Marc Diefenbruch, Elke Heck, Jorg Hintelmann, and Bruno Müller-Clostermann. Performance evaluation of SDL systems adjunct by queueing models. In R. Braek and A. Sarma, editors, *SDL'95 With MSC in CASE, Proceedings of the Seventh SDL Forum*, pages 231–242, Oslo, Norway, September 1995. Elsevier.

[Die97] Marc Diefenbruch. Queuing SDL: A language for the functional and quantitative specification of distributed systems. Technical report, Universität Essen, Germany, February 1997.

[Dil89] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer-Verlag, 1989.

[DKRT97] Pedro R. D'Argenio, Joost-Peter Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! Technical Report CTIT 97-03, Univ. of Twente, 1997.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In *DIMACS Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*. Springer Verlag, October 1995.

[Dou98] Bruce P. Douglass. *Real-Time* UML*: Developing Efficient Objects for Embedded Systems.* Object Technology Series. Addison Wesley, 1998.

[Dou99] Bruce P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.* Object Technology Series. Addison Wesley, 1999.

[EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

[Eng96] John English. *Ada 95: The Craft of Object-Oriented Programming.* Prentice Hall, October 1996.

[ETS00] ETSI/MTS. Methods for testing and specification(MTS); the tree and tabular combined notation version 3. ETSI recommendation DES/MTS–63–(1–3), November 2000.

[FDT95] Joachim Fischer, E. Dimitrov, and U. Taubert. Analysis and formal verification of SDL-92. specification using extended petri nets. Technical report, Humboldt Universität, Berlin, 1995.

[FG97] Hans Fleischhack and Bernd Grahlmann. A compositional petri net semantics for SDL. Technical Report HIB 18/97, Universität Hildesheim, 1997.

[GGP99] Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. Towards a new formal SDL semantics based on abstract state machines. In Rachida Dssouli, Gregor. v. Bochmann, and Yair Lahav, editors, *SDL '99 – The next milenium*, pages 171–190. Elsevier, 1999.

[God91] Jens C. Godskesen. An operational semantic model for basic SDL. Technical Report TFL-RR 1991-2, Tele Danmark Research, August 1991.

[GPR93] J. Grabowski, P.Graubmann, and E. Rudolph. Towards a petri net based semantics definition for message sequence charts. In O. Faergemand and A. Sarma, editors, *SDL'93 Using Objects. Proceedings of the 6th SDL Forum*, Amsterdam, 1993. Elsevier Science.

[GR89] Adele Goldberg and David Robson. *Smalltalk 80 : The Language.* Addison-Wesley, June 1989.

[Gro89] Roland Groz. *Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations.* PhD thesis, Université de Rennes I, Janvier 1989.

[Gur88] Yuri Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Yuri Gurevich. 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, EECS Department, 1997.

[GvdP96] J.F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*. Springer Verlag, 1996.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, (8):231–274, 1987.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, 1996.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1+2):110–22, 1997.

[HKN01] Dieter Hogrefe, Beat Koch, and Helmut Neukirchen. Some implications of msc, sdl and ttcn time extensions for computer-aided test generation. In *Proceedings of the 10th SDL-Forum*, LNCS, Copenhagen, June 2001. Springer Verlag.

[HKPV98] Thomas A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, (58):94–124, 1998.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *Proceedings of ICALP'92*, volume 623 of *LNCS*. Springer Verlag, 1992.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[HP88] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1988.

[HPY96] Gerard J. Holzmann, Doron Peled, and M. Yannakakis. On nested depth first search. In AMS, editor, *Second SPIN Workshop*, pages 23–32, 1996.

[HS96] K. Havelund and N. Shankar. Experiments in theorem prooving and model checking for protocol verification. In M-C. Glaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advance in Formal Methods*, volume 1051 of *LNCS*. Springer Verlag, 1996.

[ISO89a] ISO/IEC. ESTELLE– a formal description technique based on an extended state transition model. ISO 9074:1989, International Organization for Standardization – Information processing systems – Open Systems Interconnection, 1989.

[ISO89b] ISO/IEC. LOTOS– a formal description technique based on the temporal ordering of observational behavior. ISO 8807:1989, International Organization for Standardization – Information processing systems – Open Systems Interconnection, 1989.

[ISO92] ISO/IEC. Conformance testing methodology and framework. part 3: The tree and tabular combined notation (ttcn). ISO/IEC 9646-3, International Organization for Standardization – Information processing systems – Open Systems Interconnection, 1992.

[ISO96] ISO/IEC. Vienna Development Method – Specification Language – Part 1: Base language. ISO/IEC 13817-1, International Organization for Standardization – Information technology – Programming languages, their environments and system software interfaces, December 1996.

[IT97] ITU-T. Supplement 1 to recommendation z.100 - SDL + methodology: Use of MSC and SDL (with ASN.1), May 1997.

[IT99a] ITU-T. Languages for telecommunications applications – Message Sequence Charts (MSC). ITU-T Revised Recommendation Z.120, November 1999.

[IT99b] ITU-T. Languages for telecommunications applications – Specification and Description Language (SDL). ITU-T Revised Recommendation Z.100, 1999.

[IT99c] ITU-T. SDL formal definition. ITU-T Revised Recommendation Z.100 – Annex F, 1999.

[Kan92] Krishna Kant. *Introduction To Computer System Performance Evaluation.* McGraw-Hill, 1992.

[KJG99] Alain Kerbrat, Thierry Jéron, and Roland Groz. Automated test generation from SDL specifications. In R. Dssouli, G.V. Bochmann, and Y. Lahav, editors, *Proceedings of the 9th SDL Forum*, pages 135–51, Montreal, Canada, 1999. Elsevier Science.

[KM95] Niels Ferdinand Karstensen and Simon Mork. Duration calculus semantics for SDL. Master's thesis, Technical University of Denmark, Lyngby, Denmark, 1995.

[KO99] Alain Kerbrat and Iulian Ober. Automated test generation from SDL/UML specifications. In *The 12th International Software Quality Week*, San Jose, California, May 1999.

[KPSY93] Yonit Kesten, Amir Pnueli, Joseph Sifakis, and Sergio Yovine. Integration graphs: A class of decidable hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, number 736 in LNCS. Springer-Verlag, 1993.

[KRPO93] Mark H. Klein, Thomas Ralya, Bill Pollak, and Ray Obenza. *A Practitioner's Handbook for Real-Time Analysis : Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer International Series in Engineering. Kluwer Academic Publishers, 1993.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[LL93] P.B. Ladkin and S. Leue. What do message sequence charts mean? In R.L. Tenney, P.D. Amer, and M. Uyar, editors, *Proceedings of the 6th International Conference on Formal Description Techniques*, Amsterdam, 1993. North-Holland.

[Loc98] Douglass Locke. Fundamentals of real-time. OMG document realtime/98-05-03, May 1998. Tutorial presented at the OMG Technical Committee Meeting.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Springer International Journal on Software Tools for Technology Transfer*, (1(1+2)), 1997.

[Mam00] Zoubir Mammeri. *SDL. Modélisation de protocoles et systèmes réactifs.* Hermes Science, 2000. In French.

[Mat96] R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In Z. Brezocnik and T. Kapus, editors, *Proceedings of COST 247: International Workshop on Applied Formal Methods in System Design.* Technical Report, Univ. of Maribor, Slovenia, 1996. Also available as INRIA Technical Report No. 2965.

[Mey95] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1995.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[MF76] P.M. Merlin and D.J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communications*, Com-24(9):1036–1043, September 1976.

[MGHS96] Simon Mork, Jens C. Godskesen, Michael R. Hansen, and Robin Sharp. A timed semantics for SDL. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of FORTE/PSTV'96*, pages 295–309, Kaiserslautern, Germany, October 1996. Chapman and Hall.

[Mil80] Robin Milner. A calculus of communicating systems. volume 92 of *LNCS*. Springer Verlag, 1980.

[Min99] Marius Minea. *Partial Order Reduction for Verification of Timed Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1999. Available as technical report CMU-CS-00-102.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*. Springer Verlag, June 1991.

[MP00] Anca Muscholl and Doron Peled. Analyzing message sequence charts. In Université de Grenoble, editor, *SDL and MSC. Proceedings of SAM'2000*, 2000.

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[MR96] S. Mauw and M.A. Reniers. The formalization of mesage sequence charts. *Computer Networks and ISDN Systems*, 28(12), 1996.

[MT00] Andreas Mitschele-Thiel. *Systems Engineering with SDL. Developing Performance Critical Communication Systems*. J. Wiley, 2000.

[Nic92] X. Nicollin. *ATP: une algèbre pour la spécification et l'analyse des systèmes temps réel*. PhD thesis, INP Grenoble, 1992. In French.

[NOSY93] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems II*, volume 736 of *LNCS*. Springer Verlag, 1993.

[NS91] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*, pages 526–548. Springer Verlag, June 1991.

[NSY91] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From atp to timed graphs and hybrid systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*. Springer Verlag, June 1991.

[Obe99] Iulian Ober. Extending SDL with timed automata concepts. Technical report, VERILOG, 1999.

[Obe00] Iulian Ober. UML and the tasks specific to real-time development. Position Paper in the UML'2000 Workshop on Formal Design Techniques for Real-Time UML, October 2000.

[OFMP+94] Anders Olsen, Ove Faergemand, Birger Moller-Pedersen, Rick Reed, and J.R.W. Smith. *Systems Engineering Using SDL–92*. North Holland - Elsevier Science, 1994.

[OK99] Iulian Ober and Alain Kerbrat. Specification and execution of tests using tMSC. In J. Wu, S.T. Chanson, and Q. Gao, editors, *Proceedings of FORTE/PSTV'99*, pages 453–468. Kluwer Academic Publishers, 1999.

[OK01] Iulian Ober and Alain Kerbrat. Verification of quantitative temporal properties of SDL specifications. In *Proceedings of the 10th SDL Forum*, LNCS, Copenhagen, 2001. Springer Verlag.

[OMG99] OMG. Unified modeling language specification, v. 1.3. OMG ducument ad/99-06-09, June 1999.

[OS99] Iulian Ober and Ileana Stan. On the concurrent object model of UML. In *Proceedings of EUROPAR'99*, LNCS. Springer Verlag, 1999.

[OSY94] Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. Using abstractions for the verification of linear hybrid systems. In *Proceedings of 6th Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, California, July 1994. Springer-Verlag.

[Pag96] Florence Pagani. Partial orders and verification of real-time systems. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 327–46, Uppsala,Sweden, September 1996. Springer-Verlag.

[Pap92] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object Oriented Programming*. PhD thesis, Université de Genève, 1992.

[Per90] Jean-Paul Perez. *Systèmes temps réel*. Informatique Industrielle. Dunod, 1990. In French.

[Pet81] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1981.

[PL93] P.B.Ladkin and S. Leue. Interpreting message sequence charts. Technical Report TR101, Department of Computing Science, University of Stirling, 1993.

[PMR+00] Sanjoy Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. The rmtp-ii protocol. IETF Draft draft-whetten-rmtp-ii-00, 2000.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*. IEEE, 1977.

[QS82] J.P. Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*. Springer Verlag, 1982.

[Ram74] Chander Ramachandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, MIT, 1974.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, December 1998.

[Rou98] Jean-Luc Roux. SDL performance analysis with *Object*GEODE. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in SDL and MSC*, Erlangen, Germany, February 1998. Friedrich-Alexander Universität, Erlangen-Nürnberg.

[Set96] Ravi Sethi. *Programming Languages : Concepts and Constructs*. Addison-Wesley, 2nd edition, 1996.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley Professional Computing. John Wiley, 1994.

[Sha95] Alan Shaw. Reasoning about time in higher-level language software. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 16, pages 374–406. Prentice Hall, 1995.

[Sif77] Joseph Sifakis. Use of Petri nets for performance evaluation. In E. Beilner and E. Gelenbe, editors, *Measuring, Modeling and Evaluating Computer Systems*, pages 75–93. North Holland, 1977.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. whitepaper, ObjecTime Ltd., March 1998.

[SSR89] Roberto Saracco, J.R.W. Smith, and Rick Reed. *Telecommunications Systems Engineering Using SDL*. North Holland - Elsevier Science, 1989.

[Sta01] Ileana Stan. *Harminization of Modeling Languages with Object Oriented Extensions and Executable Semantics*. PhD thesis, Institut National Polytechnique de Toulouse, Avril 2001.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, July 1997.

[sWG00] SpaceWire Working Group. SpaceWire: Serial point-to-point links. European Space Agency document UoD-DICE-TN-9201, Issue D, May 2000. http://www.estec.esa.nl/tech/spacewire.

[tadwg00] Real time analysis and design working group. Response to the omg rfp for schedulability, performance and time, v.1.0. OMG Document ad/00-08-04, August 2000.

[Tar72] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[TEL00a] TELELOGIC A.B., Malmö, Sweden. ObjectGEODE *4.1 Reference Manuals*, 2000.

[TEL00b] TELELOGIC A.B., Malmö, Sweden. T*elelogic* TAU *SDL Suite Reference Manuals*, 2000.

[Tri98] Stavros Tripakis. *The Formal Analysis of Timed Systems in Practice.* PhD thesis, Univertité Joseph Fourier, Grenoble, France, 1998.

[Weg87] Peter Wegner. Dimensions of object based language design. In *Proceedings of OOPSLA'87*, volume 22 of *ACM SIGPLAN Notices*, pages 168–182, Orlando, Florida, October 1987.

[WG97] Thomas Walter and Jens Grabowski. Real-time ttcn for testing real-time and multimedia systems. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[WPT99] Brian Whetten, Sanjoy Paul, and Gursel Taskale. Rmtp-ii overview. White paper, Talarian Corp., September 1999.

[Yov97] Sergio Yovine. KRONOS: a verification tool for real-time systems. *Springer International Journal on Software Tools for Technology Transfer*, (1(1+2)), 1997.

Appendix A

List of abbreviations

ASM	Abstract State Machines
DBM	Difference Bounds Matrix
GOAL	GEODE Observer Automata Language
HMSC	High-Level Message Sequence Charts
ITU–T	International Telecommunication Union, Telecommunication Standardization Sector
LTS	Labeled Transition System
MSC	Message Sequence Charts
SAM	SDL Abstract Machine
SDL	Specification and Description Language
TA	Timed Automata
TBA	Timed Büchi Automata
TPA	Timed Property Automata
UML	Unified Modeling Language
Z.100	ITU–T Recommendation Z.100 – Specification and Description Language
Z.120	ITU–T Recommendation Z.120 – Message Sequence Charts

Appendix B

Proofs

Proof of lemma 8.1. We prove the equality in two steps:

1. $\boxed{(q, S') \subseteq \mathsf{time\text{-}succ}((q, S))}$

   Let $\mathbf{v} \in S'$. This implies the following:

   (a) $\mathbf{v} \in \nearrow S$, and

   (b) $\forall e_i$ *eager* transition originating in q, $\mathbf{v} \in \mathsf{restrict\text{-}eager}(S, \zeta_i)$, and

   (c) $\forall e_i$ *delayable* transition originating in q, $\mathbf{v} \in \mathsf{restrict\text{-}delayable}(S, \zeta_i)$.

   We aim to prove that $\exists \delta \in \mathbb{R}_+$ such that $\mathbf{v} - \delta \in S$ and the time progress conditions are met for the transition $(q, \mathbf{v} - \delta) \xrightarrow{\delta} (q, \mathbf{v})$.

   From **(a)**, we have that $\exists \delta_0 \in \mathbb{R}_+ .\ \mathbf{v} - \delta_0 \in S$. In the following we will define δ_1,...,δ_l, each δ_i corresponding to a transition e_i, such that the following conditions (the time progress conditions imposed by e_i) are met by each δ_i:

   (i) $\mathbf{v} - \delta_i \in S$.

   (ii) if e_i is *eager*, then $\forall \delta' \in (0, \delta_i]$, $\mathbf{v} - \delta' \notin \zeta_i$.

   (iii) if e_i is *delayable*, then $\forall \delta', \delta''$ such that $0 \leq \delta'' < \delta' \leq \delta_i$, $(\mathbf{v} - \delta' \in \zeta_i \Rightarrow \mathbf{v} - \delta'' \in \zeta_i)$.

   We define δ_1,...,δ_l as follows:

   - if e_i is *lazy*: $\delta_i = \delta_0$,
   - if e_i is *eager*: from **(b)** we have that $\mathbf{v} \in \mathsf{restrict\text{-}eager}(S, \zeta_i)$. From the definition of restrict-eager, the following three cases are possible:

     $\underline{\mathbf{v} \in S \cap \zeta_i}$ In this case, we choose $\delta_i = 0$. It is easy to see that δ_i satisfies **(i)** and **(ii)** (e_i being *eager*, **(iii)** does not concern δ_i).

     $\underline{\mathbf{v} \in (\nearrow (S \cap (\swarrow \zeta_i \setminus \zeta_i))) \cap (\swarrow \zeta_i \setminus \mathsf{open\text{-}inf}(\zeta_i))}$ In this case $\mathbf{v} \in (\nearrow (S \cap (\swarrow \zeta_i \setminus \zeta_i)))$ $\Rightarrow \mathbf{v} \in \nearrow S$. We choose δ_i arbitrary such that $\mathbf{v} - \delta_i \in S$ (i.e. satisfying **(i)**). We prove that δ_i satisfies **(ii)**.
     $\mathbf{v} \in (\swarrow \zeta_i \setminus \mathsf{open\text{-}inf}(\zeta_i)) \Rightarrow \mathbf{v} \in \swarrow \zeta_i$ and $\mathbf{v} \notin \mathsf{open\text{-}inf}(\zeta_i)$. The following two cases are possible:

- $\mathbf{v} \notin \zeta_i$. Since $\mathbf{v} \in_{\swarrow} \zeta_i$ and ζ_i is a convex polyhedron[1], it is easy to see that this implies $\forall \delta' \in \mathbb{R}_+$, $\mathbf{v} - \delta' \notin \zeta_i$, so we have **(ii)**.
- $\mathbf{v} \in \zeta_i$. But $\mathbf{v} \notin \mathsf{open\text{-}inf}(\zeta_i)$, and from the definition of open-inf this implies that $\forall \delta' \in \mathbb{R}_+$, $\mathbf{v} - \delta' \notin \zeta_i$, so we have **(ii)**.

$\underline{\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i)}$ We have $\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i) \Rightarrow \mathbf{v} \in \nearrow S$. We choose δ_i arbitrary such that $\mathbf{v} - \delta_i \in S$ (i.e. satisfying **(i)**). We prove that δ_i satisfies **(ii)**.
$\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i) \Rightarrow \forall \delta' \in \mathbb{R}_+ .\ \mathbf{v} - \delta' \notin_{\swarrow} \zeta_i$, whence $\mathbf{v} - \delta' \notin \zeta_i$, whence **(ii)**.

- if e_i is *delayable*: from **(c)** we have that $\mathbf{v} \in \mathsf{restrict\text{-}delayable}(S, \zeta_i)$. From the definition of restrict-delayable, the following two cases are possible:

$\underline{\mathbf{v} \in (\nearrow (S \cap (\swarrow \zeta_i)) \cap (\swarrow \zeta_i)}$ In this case, $\mathbf{v} \in \nearrow (S \cap (\swarrow \zeta_i)) \Rightarrow \mathbf{v} \in \nearrow S$. We choose δ_i arbitrarily such that $\mathbf{v} - \delta_i \in S$ (i.e. satisfying **(i)**). We prove that δ_i satisfies **(iii)** (e_i being *delayable*, δ_i is not concerned by **(ii)**).
Since $\mathbf{v} \in_{\swarrow} \zeta_i$, it follows that $\exists \delta''' \in \mathbb{R}_+$ such that $\mathbf{v} + \delta''' \in \zeta_i$. Assume that **(iii)** is not satisfied. Then, $\exists \delta', \delta''.\ 0 \leq \delta'' < \delta' \leq \delta_i$ such that $\mathbf{v} - \delta' \in \zeta_i$ and $\mathbf{v} - \delta'' \notin \zeta_i$.
The relative position of $\mathbf{v} - \delta'$,$\mathbf{v} - \delta''$ and $\mathbf{v} + \delta'''$ imply that the polyhedron ζ_i is not convex, which contradicts the definition of timed automata transitions. We conclude that **(iii)** is satisfied.

$\underline{\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i)}$ We have $\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i) \Rightarrow \mathbf{v} \in \nearrow S$. We choose δ_i arbitrarily such that $\mathbf{v} - \delta_i \in S$ (i.e. satisfying **(i)**). Similarly to the *eager* case, we have: $\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i) \Rightarrow \forall \delta' \in \mathbb{R}_+ .\ \mathbf{v} - \delta' \notin_{\swarrow} \zeta_i$, whence $\mathbf{v} - \delta' \notin \zeta_i$. This means the the premise of the implication from **(iii)** is false, and therefore **(iii)** is satisfied.

As each δ_i satisfies the conditions of **(i)**, **(ii)** and **(iii)** for the transition e_i, it is easy to see that by taking $\delta = min\{\delta_0, \delta_1, ..., \delta_l\}$, the number δ satisfies the time progress conditions for all transitions $e_1, ..., e_l$.

Therefore, we conclude that $\exists \delta \in \mathbb{R}_+$ such that $(q, \mathbf{v} - \delta) \in (q, S)$ and $(q, \mathbf{v} - \delta) \xrightarrow{\delta} (q, \mathbf{v})$, which implies that $(q, \mathbf{v}) \in \mathsf{time\text{-}succ}((q, S))$.

2. $\boxed{\mathsf{time\text{-}succ}((q, S)) \subseteq (q, S')}$

Let $(q, \mathbf{v}) \in \mathsf{time\text{-}succ}((q, S))$. This implies that $\exists \delta.\ (q, \mathbf{v} - \delta) \xrightarrow{\delta} (q, \mathbf{v})$, which by definition means that:

(a) $\exists \delta$ such that $\mathbf{v} - \delta \in S$, and

(b) $\forall e_i$ *eager* transition starting from q, $\forall \delta' \in (0, \delta]$, $\mathbf{v} - \delta' \notin \zeta_i$, and

(c) $\forall e_i$ *delayable* transition starting from q, $\forall \delta'$, δ'' such that $0 \leq \delta'' < \delta' \leq \delta$, $(\mathbf{v} - \delta' \in \zeta_i \ \Rightarrow \ \mathbf{v} - \delta'' \in \zeta_i)$.

From **(a)**, it follows that $\mathbf{v} \in \nearrow S$. In order to prove that $\mathbf{v} \in S'$, we show that for every transition e_i starting from q:

(i) if e_i is *eager*, then $\mathbf{v} \in \mathsf{restrict\text{-}eager}(S, \zeta_i)$,

[1] If ζ is convex, then $\nearrow \zeta \cap \swarrow \zeta = \zeta$.

(ii) if e_i is *delayable*, then $\mathbf{v} \in \mathsf{restrict\text{-}delayable}(S, \zeta_i)$,

If e_i is *eager*, we distinguish the following cases:

- $\underline{\mathbf{v} - \delta \in \zeta_i}$.

  If $\delta \neq 0$, by taking $\delta' = \delta$ in **(b)**, we get $\mathbf{v} - \delta \notin \zeta_i$, which contradicts the hypothesis. Therefore, $\delta = 0$, and $\mathbf{v} = \mathbf{v} - \delta \in S \cap \zeta_i$, which by definition implies **(i)**.
- $\underline{\mathbf{v} - \delta \in (\swarrow \zeta_i \setminus \zeta_i)}$.

  Then $\mathbf{v} - \delta \in S \cap (\swarrow \zeta_i \setminus \zeta_i)$, which implies $\mathbf{v} \in \nearrow (S \cap (\swarrow \zeta_i \setminus \zeta_i))$. We further need to prove that $\mathbf{v} \in (\swarrow \zeta_i \setminus \mathsf{open\text{-}inf}(\zeta_i))$, which we do in two steps:
  - $\mathbf{v} - \delta \in_{\swarrow} \zeta_i \setminus \zeta_i \Rightarrow \mathbf{v} - \delta \in_{\swarrow} \zeta_i \Rightarrow \exists \delta''$ such that $(\mathbf{v} - \delta) + \delta'' \in \zeta_i$.
    Since $(\mathbf{v} - \delta) \notin \zeta_i$, we have that $\delta'' \neq 0$. Assume that $0 < \delta'' < \delta$. Then $(\mathbf{v} - \delta) + \delta'' = \mathbf{v} - (\delta - \delta'') \in \zeta_i$ and $0 < (\delta - \delta'') < \delta$ which contradicts **(b)**.
    The above considerations imply $\delta'' \geq \delta$. But since $\mathbf{v} + (\delta'' - \delta) \in \zeta_i$, this implies $\mathbf{v} \in_{\swarrow} \zeta_i$.
  - To prove that $\mathbf{v} \notin \mathsf{open\text{-}inf}(\zeta_i)$, we proceed by negation.
    Assume that $\mathbf{v} \in \mathsf{open\text{-}inf}(\zeta_i)$. Then, by definition, $\exists \delta' > 0$. $\mathbf{v} - \delta' \in \zeta_i$. As $\mathbf{v} \in \zeta_i$ and $\mathbf{v} - \delta \notin \zeta_i$, from the convexity of ζ_i it results that $0 < \delta' < \delta$. However, this contradicts **(b)**, and therefore $\mathbf{v} \notin \mathsf{open\text{-}inf}(\zeta_i)$ holds.

  We conclude that in this case $\mathbf{v} \in \nearrow (S \cap (\swarrow \zeta_i \setminus \zeta_i)) \cap (\swarrow \zeta_i \setminus \mathsf{open\text{-}inf}(\zeta_i))$, which by definition implies **(i)**.
- $\underline{\mathbf{v} - \delta \notin_{\swarrow} \zeta_i}$.

  Since $\mathbf{v} - \delta \in S$ and $\mathbf{v} - \delta \notin_{\swarrow} \zeta_i$, it results that $\mathbf{v} - \delta \in (S \setminus \swarrow \zeta_i)$, which implies that $\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i)$. By definition, this implies **(i)**.

If e_i is *delayable*, we distinguish the following cases:

- $\underline{\mathbf{v} - \delta \in_{\swarrow} \zeta_i}$.

  $\mathbf{v} - \delta \in_{\swarrow} \zeta_i$ and $\mathbf{v} - \delta \in S$ imply that $\mathbf{v} \in \nearrow (S \cap \swarrow \zeta_i)$. We further prove that $\mathbf{v} \in_{\swarrow} \zeta_i$. We distinguish two cases:
  - $\mathbf{v} - \delta \in \zeta_i$ or $\exists \delta'$, $0 \leq \delta' < \delta$ such that $\mathbf{v} - \delta' \in \zeta_i$. In this case, from **(c)** by choosing $\delta'' = 0$, we have $\mathbf{v} \in \zeta_i$, which implies $\mathbf{v} \in_{\swarrow} \zeta_i$.
  - $\forall \delta' \in [0, \delta]$, $\mathbf{v} - \delta' \notin \zeta_i$.
    But $\mathbf{v} - \delta \in_{\swarrow} \zeta_i$ implies that $\exists \delta''$ such that $(\mathbf{v} - \delta) + \delta'' \in \zeta_i$. The above hypothesis implies however that $\delta'' > \delta$, and therefore $\mathbf{v} \in_{\swarrow} \zeta_i$.

  The above considerations lead to $\mathbf{v} \in \nearrow (S \cap \swarrow \zeta_i) \cap \swarrow \zeta_i$, which by definition implies **(ii)**.
- $\underline{\mathbf{v} - \delta \notin_{\swarrow} \zeta_i}$.

  Since $\mathbf{v} - \delta \in S$ and $\mathbf{v} - \delta \notin_{\swarrow} \zeta_i$, it results that $\mathbf{v} - \delta \in (S \setminus \swarrow \zeta_i)$, which implies that $\mathbf{v} \in \nearrow (S \setminus \swarrow \zeta_i)$. By definition, this implies **(ii)**.

To conclude, we have proved that $(q, \mathbf{v}) \in \mathsf{time\text{-}succ}((q, S)) \Rightarrow \mathbf{v} \in S'$, and therefore the sought inclusion holds.

■