THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue par :
Elie FARES

le mardi 12 mars 2013

Titre :

Real-Time Systems Refinement
Application to the Verification of Web Services

École doctorale et discipline ou spécialité :

ED MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

- M. Jean-Paul BODEVEIX
- M. Mamoun FILALI

Jury :

- M. Hassan Mountassir, Université de Franche-Comté - Rapporteur
- M. Pascal Poizat, Université Paris Ouest - Rapporteur
- M. Yamine Ait-Ameur, INP, Université de Toulouse - Examinateur
- M. Christian Attiogbé, Université de Nantes - Examinateur
- M. Jean Paul Bodeveix, UPS, Université de Toulouse - Directeur de thèse
- M. Mamoun Filali-Amine, CNRS, Université de Toulouse - CoDirecteur

To all who supported me

Thanks

This is a life where we meet so many people. Some people may come and go, but others will stay with us forever. In these last four years, even though I spent most of my time in a small office, I came to meet and know certain people who I would like to thank in this note.

I would like at first to thank my advisors, Jean-Paul Bodeveix and Mamoun Filali with whom I took my first steps in the research world during my Masters 2 Internship. Our work together is now crowned with the end of my PhD in which their advices, help, availability and devotion made this quest possible. For this I am and will be always grateful.

I would like also to thank Hassan Mountassir, Pascal Poizat, Yamine Ait-Ameur and Christian Attiogbé for accepting to be my on jury board. Special thanks especially to Hassan Mountassir and Pascal Poizat for their reports. Their remarks have surely made this work even better.

A thank you to Antoine Leger, the coordinator of the ITEmIS project in which I have participated. I thank him for his patience during the deadlines and his constant motivation towards reaching the goals of the project. I have shared a good professional experience with Antoine.

A big thank you from the heart to my family, father, mother and brother for believing in me and pushing me forward towards reaching my dreams. I am blessed for their presence in my life.

I also thank my girl-friend Karen, who has been with me, hand in hand, all the time. She knew how to deal with the ups and downs of my PhD. Karen gave me the strength I needed to finish my PhD.

I finally thank all my friends, IRIT colleagues and especially the ACADIE team members, Bertrand, Petra, Pascal, Selma, Iulia, Nadia, Manuel... . Things would not have been the same without them. I thank Jean-Baptiste as well for the recent conversations concerning my future career.

Last but not least, I thank God, the one who was pulling the strings all the way.

Chapter

Résumé

Aujourd'hui, et avec la rapidité des progrès technologiques à laquelle nous assistons, nous pouvons dire que nous nous seront bientôt gouvernés par des machines. Peut-être pas comme dans le film *terminator*, mais par notre dépendance dans la vie quotidienne à l'égard du bon fonctionnement des systèmes automatisés. Ces systèmes sont de plus en plus complexes et sont utilisés dans de nombreux domaines tels que les télécommunications, l'avionique, de l'automobile, l'aérospatial, le ferroviaire, le médical, la finance, les services Web et d'autres domaines connexes. Par conséquent, un degré élevé de sécurité de ces systèmes est devenu une condition indispensable à leurs bon fonctionnement.

En fait, les défaillances logicielles peuvent entraîner des pertes et dégâts importants que ce soit financieres ou alors de vies humaines. Selon Computerworld WASHINGTON dans un rapport qui remonte à 2002, "*les defaillances logiciels coûtent à l'économie américaine environ 59,5 milliards de dollars chaque année, dont plus de la moitié du coût supporté par les utilisateurs finaux et le reste par les développeurs et les fournisseurs*".

Plusieurs événements dans le passé ont montré la multitude de dégâts que les erreurs logicielles peuvent provoquer. Dans certains cas, les dommages ont été décrits comme catastrophiques.

Rappelons-nous la panne bien connue de Northeast en 2003, où une panne d'électricité massive s'est produit le 14 Août 2003 et a laissé de vastes régions du nord-est et du Midwest des États-Unis et de l'Ontario au Canada sans courant. En raison d'une erreur de programmation, de multiples systèmes ont tenté d'accéder simultanément aux mêmes informations. Ce chaos a provoqué un dysfonctionnement de l'alarme par lequel le logiciel en question a envoyé un signal d'occupation à tous les systèmes. Cette bévue technologique coûteuse a conduit à une file d'attente d'événements non traités et par conséquent a ralenti la performance qui ensuite a causé le problème. En conséquence, les pertes ont été d'environ 7 à 10 milliards de dollars.

Un autre exemple notoire est le crash d'Ariane 5, le lanceur européen de satellites dont l'Agence spatiale européenne a passé dix années à créer avec un coût total de huit milliards de dollars. Le but ultime d'Ariane 5 devait donner à l'Europe un rôle pionnier dans le domaine de l'espace. Malheureusement, le 4 Juin 1996, Ariane 5 a explosé dans le ciel de Guienna française seulement 36,7 secondes après son lancement. L'accident a été le résultat d'une conversion 64 bits à virgule flottante en un entier

16 bits. En effet, le nombre était trop grand et a entraîné une erreur de débordement qui a bloqué les ordinateurs principaux et de sauvegarde. Les pertes ont été de 3650 jours de travail ainsi que de 500 millions de dollars en coûts de satellites et de 8 milliards de dollars de coûts de production.

Tous ces événements regrettables qui ont causé d'immenses pertes de temps, de travail, de vie humaine, d'argents, sont une preuve qu'il est très important d'avoir une phase de vérification et de validation des systèmes critiques avant leur mise en oeuvre définitive dans le monde réel. Afin de remplir cette obligation impérative, les méthodes formelles ont été proposées.

0.1 Méthodes Formelles

Les méthodes formelles sont des techniques, languages et outils basés sur les mathématiques qui sont utilisés pour la spécification et la vérification des logiciels informatiques et du matériel. Les méthodes formelles sont utilisées pour donner une garantie elevée de la spécification d'un système. Cela augmente la confiance en son bon fonctionnement en révélant les bugs qui pourraient autrement passer inaperçus.

0.1.1 Spécification des Systèmes

Les méthodes formelles peuvent être utilisées dans la phase de spécification du système lorsque les exigences d'un système sont décrits.

Une spécification est un modèle formel qui représente les éléments statiques et dynamiques d'un système. Les aspects statiques sont représentés par des états alors que les aspects dynamiques sont représentés par des actions (transitions entre les différents états). Les réseaux de Petri [91], CSP [96], CCS [83] et la Méthode-B [7] sont tous des exemples de langages de spécification formelle à base de modèles.

0.1.2 Vérification des Systèmes

Une fois une spécification formelle du système donnée, la spécification peut être validée par la preuve de certaines propriétés. Plusieurs techniques de vérification existent, qui varient entre des techniques automatiques ou des techniques manuelles. Les techniques de vérification du système se répartissent en quatre catégories générales: la preuve de théorèmes, le model checking, l'interprétation abstraite et le test de logiciels.

La preuve de théorèmes consiste à prouver la satisfaction des propriétés sur les systèmes au moyen de règles d'inférence et d'induction. Bien que certaines de ces preuves soient manuelles, les autres sont soit automatique ou interactive. Il existe plusieurs outils qui offrent la possibilité de développer des preuves automatiques ou interactives. Dans un outil de preuves automatisé, une preuve automatique peut être construite à partir d'une description du système, un ensemble d'axiomes logiques, et un ensemble de règles d'inférences. Des exemples de ces outils sont les Satisfiability Modulo Théories (Solveurs SMT) [46] (comme STP [51]) où le

problème est de déterminer si une formule logique peut être satisfaite sur une combinaison de théories exprimée en logique du premier ordre avec égalité. Contrairement à une preuve automatique, un outil de preuve interactive nécessite généralement une intervention humaine. Des exemples de ces outils sont Coq [102], Isabelle [88] et B etc.

Une deuxième technique de vérification est l'interprétation abstraite. L'interprétation abstraite [42] a été fondée par Patrick Cousot et Radhia Cousot vers la fin des années 1970. C'est une méthode qui repose sur le remplacement d'un domaine infini de valeurs concrètes d'un système par un certain nombre de domaines abstraits finis où les opérations initiales (concrétes) sont redéfinis. Des propriétés, telles que les bornes des intervalles et l'absence d'accès à des pointeurs nuls, sont ensuite vérifiées sur le modèle abstrait.

Une autre technique consiste en des tests de logiciels. Le test de logiciel est le processus de l'exécution du code réel d'un programme en spécifiant ses paramètres et en vérifiant si il se comporte correctement. Le test permet une vérification de la conformité du système par rapport à sa définition et à examiner si le logiciel répond aux exigences. Le problème de cette technique est la nécessité de générer des cas de tests suffisants afin d'avoir une plus grande confiance dans le produit. Toutefois, étant donné le nombre de cas de test tend à être infini, les tests peuvent pas être exhaustive (au contraire du model checking: voir ci-dessous). Ainsi, en faisant du test, on ne peut pas prouver qu'un produit se comporte correctement dans toutes les conditions, mais plutôt on peut démontrer qu'il se comporte correctement sous certaines conditions.

Model checking

Le model checking est une technique de preuve automatisée dans lequel une propriété est vérifiée par rapport à un système à l'aide d'une recherche exhaustive de tous les états possibles d'une exécution du système. Formellement, le model checking se réfère au problème suivant: étant donné un modèle de système M et une propriété φ, nous examinons si la véracité de la propriété suivante lue comme M satisfait φ, ou aussi comme M est un modèle de φ.

$$M \models \varphi$$

Le modèle est généralement une abstraction du système écrit dans un formalisme basé sur les systèmes de transitions tandis que les propriétés sont des propriétés logiques écrites dans LTL [79] ou CTL [40]. La vérification d'une propriété sur le système s'effectue via les algorithmes d'exploration du model checking. Les premiers algorithmes proposés sont ceux de [40] et [94].

Les avantages de la technique du model checking est qu'elle est entièrement automatique et puisqu'elle s'agit d'une vérification exhaustive, elle garantie un degré élevé de confiance par rapport aux propriétés satisfaites. Un autre avantage du model checking est que dans le cas de non-satisfaction d'une propriété, un contre-exemple est également donné.

Cependant, l'inconvénient principal du model checking est que son application est limitée aux petits modèles. La vérification de modèle

souffre d'explosion combinatoire. Afin de combattre cet inconvénient, plusieurs solutions ont été proposées. Ces solutions se basent principalement sur quatre techniques. Les premières solutions sont des techniques d'abstraction [41] qui visent à réduire la taille du modèle. Les secondes sont des techniques de réduction d'ordre partiel [89] qui visent à réduire la taille de l'espace d'état recherché par l'algorithme du model checking. La troisième technique est celle de la vérification modulaire, où l'idée consiste à décomposer le système en sous-composants et de vérifier à part chacun des sous-composant. Enfin, la quatrième technique est celle du raffinement qui vise à construire un modèle d'une façon progressive et incrémentale.

Un outil qui implémente la technique du model checking est appelé un *model checker*. Comme exemples de model checkers, nous pouvons citer le model checker Spin [62] pour les modèles de systèmes de transitions, Tina [20] pour les reséaux de Petri Temporisés et Uppaal [73] pour les automates temporisés.

0.2 Contexte et Problèmatique

Le contexte de ce travail de thèse est le projet ANR itemis [2]. Ce projet vise à faciliter l'évolution du monde d'aujourd'hui de logiciels embarqués et de services informatiques vers un monde des services intégrés de façon transparente. Cette transition définit une nouvelle génération d'architectures orientées services (SOA) qui permet l'integration des systèmes IT et des systèmes embarqués. Ainsi, trois grands axes sont considérés à cet effet et ce sont les axes de services d'infrastructure, de l'entreprise, et de la vérification formelle. L'axe de vérification formelle aborde l'assurance de bout en bout la qualité de service et de l'exactitude de la vérification des modèles d'exécution.

Dans ce contexte, le travail nécessaire consistait à la ***vérification de modèles BPEL*** via une transformation à un langage de spécification formelle, FIACRE qui est un langage de spécification temps-réel. Les résultats de cette transformation (modèles FIACRE) peuvent être utilisés soit pour donner une sémantique formelle pour BPEL permettant ainsi l'analyse des constructions BPEL, ou à des fins de vérification (model-checking de propriétés logiques). Cependant, même si la vérification de modèle peut être envisagée pour les petits modèles BPEL (modèles FIACRE après la transformation), la vérification des modèles de grandes tailles n'est pas raisonnable. Le problème devait donc être abordé différemment, notamment en considérant la *développement incrémental* de modèles FIACRE. Cela a conduit à l'étude du ***raffinement*** de ***systèmes temporisés***.

La méthode adoptée dans ce document est montré dans la figure 1.1. La première étape est la transformation de BPEL vers FIACRE. Le résultat de cette transformation est ensuite abstrait (système abstrait) avant d'être vérifié. L'implémentation effective (système concret) du résultat de la transformation. On vérifie ensuite le raffinement entre le système concret et le système abstrait.

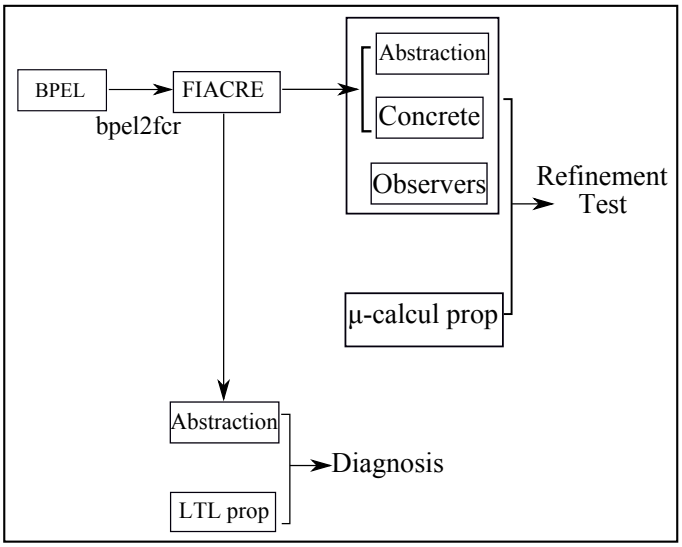


Figure 1 – *Overall Description*

0.3 Raffinement de Systèmes Temps Réels

Commençons par définir les systèmes temporisés d'une manière générale avant de définir d'une manière précise le modèle qu'on considère dans cette thèse.

0.3.1 Systèmes Temporisés

Même si les modèles classiques tels que les systèmes de transition et les réseaux de Petri peuvent exprimer le comportement d'un système, ils ne peuvent pas exprimer ses contraintes temps-réel. Cependant, le temps est une contrainte naturelle qui est explicitement présent presque partout. Des exemples de contraintes de temps peuvent être des temps d'exécution, des délais d'attente, des temps réponse, etc. Afin de répondre à ces exigences, les systèmes temporisés ont été créés afin de capturer des contraintes temporelles qui peuvent contrôler ou modifier le comportement d'un système. Donc, plusieurs modèles et logiques ont été étendus par des notions de temps.

La la plupart des travaux tournent autour les automates temporises, qui est l'un des modeles les plus étudiés pour les systèmes temporises. Les résultats les plus intéressants sur les automates temporises sont sans doute la décidabilité les propriétés d'accessibilité [15] aussi bien que la décidabilité du model-checking de TCTL (une variante temporise du CTL) [12]. En revanche, l'inclusion des traces acceptés par deux automates temporisés est indécidable [15].

Par conséquent, le développement de systèmes temporisés est sûrement plus difficile, et sa vérification devient plus complexe. Le raffinement est une technique qui permet de faciliter le développement et la vérification des systèmes temporisés.

Commençons par definir la base sémantique de notre système temporisé considéré dans cette thèfse :

Système de Transition Temporisé

Definition 0.1 (Système de Transition Temporisé TTS) *Etant donné un ensemble d'étiquettes, L_τ et soit Δ un domaine de temps, par exemple $\mathbb{R}^+$, un Système de Transition Temporisé (Timed Transition System TTS) est un tuple $\langle Q, Q^0, \rightarrow \rangle$ sachant que Q est un ensemble d'états, $Q^0 \subseteq Q$ est un ensemble d'états initiaux, et $\rightarrow$ est une relation de transitions $\subseteq Q \times (L \cup \Delta) \times Q$.*

Système de Transition Temporisé Contraint (CTTS)

Un système de transition temporisé contraint (Constrained Time Transition System CTTS) est une representation syntactique finie d'un TTS. On le définit de la manière suivante :

Definition 0.2 (Constrained Time Transition System) *Etant donné un ensemble d'étiquettes L_τ, un ensemble d'intervalles $\mathbf{I}$ défini sur un domaine de temps Δ, un CTTS [45] est définit par $\langle Q, Q^0, T, L_\tau, \rho : T \rightarrow 2^{Q \times Q}, \lambda : T \rightarrow L_\tau, \iota : T \nrightarrow \mathbf{I}, \triangleright \subseteq T \times T \rangle$ où Q est un ensemble d'états, $Q^0 \subseteq Q$ est un ensemble d'états initiaux, T est un ensemble de transitions, ρ renvoie pour chaque transition un ensemble de couples d'états (source et destination), λ associe une étiquette pour chaque transition, ι associe un intervalle de temps pour chaque transition etiquettée par un événement local, (les événements globaux ne sont pas contraints) et $\triangleright$ représente une relation de dépendance entre deux transitions.*

A chaque transition t est associée implicitement une horloge. Cette horloge est réinitialisée lorsque la transition devient tirable ou qu'une transition t' avec $t' \triangleright t$ est déclanchée. Une transition tirable peut être effectivement tirée si son horlooge est comprise entre ses bornes min et max, sans toutefois pouvoir dépasser le max.

CTTS Example Dans la figure 3.1, nous donnons un exemple de CTTS dans lequel un port local est attaché à un intervalle de temps [1,2], alors que pour le port global P n'est pas contraint. Il est également intéressant de noter que l'activation l'une des 2 transitions possibles à partir de l'état initial réinitialise l'horloge de l'autre.

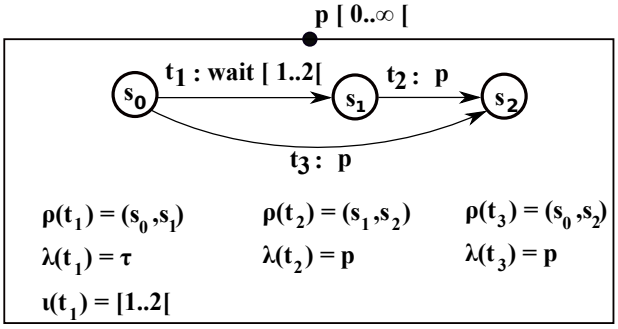


Figure 2 – *Example of CTTS*

Nous donnons un autre exemple pour illustrer la relation $\triangleright$. Dans la Fig. 3.2, nous montrons deux systèmes qui sont exactement les mêmes, sauf qu'ils diffèrent dans la manière dont le $\triangleright$ est défini entre leurs transitions. En fait, dans le premier système de la Fig. 3.2, $t_1 \triangleright t_2$, tandis que

ce n'est pas le cas dans le second système. Dans le premier système, t_2 n'est jamais tirée parce que, à l'état s_0, après exactement 1 unité de temps (u.t), seule la transition t_2 serait capable d'être tirée car elle est associée à l'intervalle de temps [1,1]. t_2 ne serait jamais tirée comme sa contrainte de temps n'est pas satisfaite. Maintenant, après le tir t_1, et comme $t_1 \triangleright t_2$, l'horloge de t_2 est remise à zéro et le même mécanisme se répète indéfinitivement. Dans le cas du second système, à 1 u.t, la transition t_1 est tirée sans reinitialiser la transition t_2. Lorsque le temps atteint 2 u.t, un choix non-déterministe entre t_1 et t_2 est donc possible.

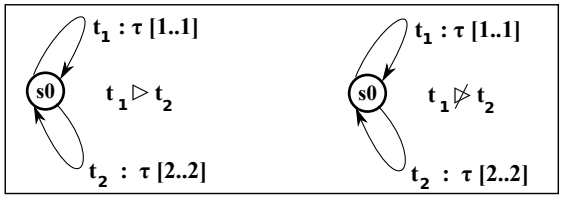


Figure 3 – *Fonction de réinitialisation d'un CTTS*

0.3.2 Raffinement et Simulation

Comme nous l'avons indiqué, le principal inconvénient du model checking est le risque d'explosion combinatoire. Le développement incrémental a été suggéré comme un moyen d'empêcher l'explosion combinatoire ou au moins pour y faire face. La vérification compositionnelle et le raffinement sont les deux techniques principales utilisées pour faire du développement incrémental. La vérification compositionnelle permet la vérification d'un système en vérifiant ses sous-composants.

Le raffinement, qui est la deuxième approche, est ce que nous considérons dans cette thèse.

Le raffinement permet de construire progressivement des spécifications correctes en le rendant plus précis. À chaque étape du processus de construction, une nouvelle spécification est dérivée d'une ancienne spécification en vérifiant que chaque nouveau niveau préserve les propriétés de l'ancien. En d'autres termes, l'ensemble du système n'est pas modélisé tout de suite, mais plutôt, il est construit progressivement par une série de modèles, où chaque nouveau modèle est censé être un raffinement de celui qui le précède. Pour pouvoir dire qu'un système raffine un autre, on a besoin de l'inclusion de leurs traces. Par contre, comme l'inclusion de traces temporisées est indécidable, on a été ramené à des conditions suffisantes qui sont les relations de simulations. Dans ce qui suit, on définit la relation de simulation faible temporisée.

Simulation Faibe Temporisée

La relation de simulation faible temporisée (Timed Weak Simulation) est définit entre deux TTS de la manière suivante :

Definition 0.3 (State-Event Timed Weak Simulation) *Etant donné un ensemble d'étiquettes L_τ et deux TTS $\mathcal{A} = \langle Q_a, Q_a^0, \rightarrow_a \rangle$ and $\mathcal{C} = \langle Q_c, Q_c^0, \rightarrow_c \rangle$ définis sur L_τ, un ensembLe de propositions P et deux fonctions de valuations $v_c : Q_c \rightarrow$*

2^P *and* $v_a : Q_a \to 2^P$*, une simulation entre eux* $\precsim$ *est la plus grande relation de sorte que :*

$$\forall q_c\ q_a, q_c \precsim q_a \Rightarrow$$

V. $v_c(q_c) = v_a(q_a)$ $(Valuations)$

E. $\forall q'_c\ e, q_c \xrightarrow{e}_c q'_c \Rightarrow \exists q'_a, q_a \xrightarrow{\tau^* e}_a q'_a \wedge q'_c \precsim q'_a$ $(Visible\ Events)$

T. $\forall q'_c, q_c \xrightarrow{\tau}_c q'_c \Rightarrow q'_c \precsim q_a$ $(\tau\ Events)$

D. $\forall q'_c\ \delta, q_c \xrightarrow{\delta}_c q'_c \Rightarrow \exists q'_a, q_a \overset{\delta^*}{\Longrightarrow}_a q'_a \wedge q'_c \precsim q'_a$ $(Delay)$

On dit que $(\mathcal{C}, v_c) \precsim (\mathcal{A}, v_a)$ si $\forall q^0_C \in Q^0_C, \exists q^0_A \in Q^0_A$ de telle sorte que $(q^0_C, q^0_A) \in R$.

Afin de préserver les propriétés du temps linéaire, des clauses supplémentaires sont ajoutés à la simulation faible temporisée. Cela conduit au Simulation Faibe Temporisée sensible au blocage (State-Event Deadlock-Sensitive (DS) Timed Weak Simulation). Nous désignons cette relation par $\precsim_{DS}$.

Definition 0.4 (Simulation Faibe Temporisée Sensible au Blocage) *Etant donné un ensemble d'étiquettes* L_τ *et deux TTS* $\mathcal{A} = \langle Q_a, Q^0_a, \to_a \rangle$ *and* $\mathcal{C} = \langle Q_c, Q^0_c, \to_c \rangle$ *définis sur* L_τ*, un ensembe de propositions* P *et deux fonctions de valuations* $v_c : Q_c \to 2^P$ *and* $v_a : Q_a \to 2^P$*, une simulation entre eux* $\precsim_{DS}$ *est la plus grande relation de telle sorte que :*

$$\forall q_c\ q_a,\ q_c \precsim_{DS} q_a \Rightarrow$$

V $\wedge$ **E** $\wedge$ **T** $\wedge$ **D**

N_C. $\mathcal{C}$ *has no-*τ*-cycle* $(Definition\ 3.3)$.

Dlk. $\forall q'_a\ e, q_a \xrightarrow{e}_a q'_a \Rightarrow \exists q'_c, q_c \xrightarrow{e}_c q'_c \wedge v_c(q'_c) = v_a(q'_a)$ $(Deadlock)$

On dit que $(C, v_c) \precsim_{DS} (A, v_a)$ si $\forall q^0_C \in Q^0_C, \exists q^0_A \in Q^0_A$ tel que $(q^0_C, q^0_A) \in R$.

La cinquième clause (**N_C**) exprime que le raffinement n'autorise pas les séquences infinies de τ transitions dans le système concret. Cela signifie qu'il doit toujours y avoir un moyen de sortir de ces cycles, via un événement visible. La dernière clause exprime le fait que le système C concret ne doit pas contenir de blocages qui n'existent pas dans le système abstrait A.

Verification de la Simulation Faibe Temporisée

Composition des Systèmes Abstrait et Concret Notre technique partage ses caractéristiques techniques avec le model-checking. En effet, la première étape consiste à composer de manière asynchrone le système abstrait avec le système concret.

Etant donné un CTTS $A = \langle Q_a, Q^0_a, T_a, L_\tau, \rho_a, \lambda_a, \iota_a, \triangleright_a \rangle$ et un CTTS $C = \langle Q_c, Q^0_c, T_c, L_\tau, \rho_c, \lambda_c, \iota_c, \triangleright_c \rangle$, les deux systèmes sont composés après avoir renommé les événements de ces deux systèmes en indexant les abstraits (resp. concret) par a (resp. c). La composition se fait donc d'une manière asynchrone (Fig. 3.12) afin d'être en mesure d'observer toutes les transitions du système concret et vérifier si elles sont simulées par le système abstrait ou non. La composition synchrone n'est pas applicable

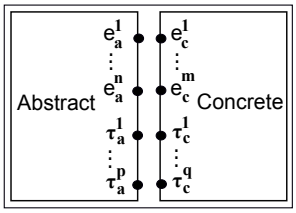


Figure 4 – *Systèmes*

parce que les transitions du système concret risquent de disparaître dans le système de produit. C'est le cas lorsqu'une transition concrète n'existe pas dans le système abstrait.

Ajout des Observateurs dans le Cadre Temporisé Pour pouvoir vérifier la simulation temporisée, nous définissons deux observateurs (Fig. 3.4 et 3.7) qui sont composés à la fois avec le système concret et le système abstrait.

1. Le premier observateur `Control Observer` consiste à observer les événements discrets des deux systèmes. Pour chaque transition du concret, l'observateur va essayer de trouver un événement correspondant dans l'abstraction se produisant en même temps.
2. Le deuxième observateur `Time Observer` modèlise l'écoulement du temps dans les deux systèmes.

Control Observer `Control Observer` est représenté dans la Fig. 3.4.

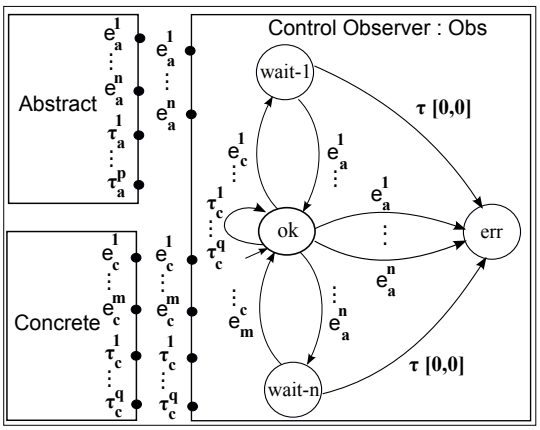


Figure 5 – *Control Observer*

A l'état initial `OK`, l'observateur se synchronise avec l'un des événements e_a^i, τ_c^i ou e_c^i. Lors de la synchronisation avec l'un des événements e_i^a l'observateur signale une erreur (`err`) comme un événement abstrait n'a pas été trouvé. Lors de la synchronisation avec n'importe lequel des τ_C^i l'observateur maintient l'état `OK`. Enfin, lors de la synchronisation avec les événements concrets e_c^i, il essaie de les faire correspondre avec les événements abstraits e_a^i. Après la reception d'événement concret e_c^i, l'observateur transite vers l'état $wait_i$ signifiant qu'il attend maintenant le passage d'un événement abstrait correspondant e_a^i. À ce stade, les scénarios suivants peuvent se produire:

- Un événement abstrait correspondant est trouvé et l'observateur transite à l'état `OK`.
- Le système abstrait viole la synchronisation du système concret et l'observateur transite à l'état `err`. Cela veut dire qu'un événement abstrait correspondant n'est pas possible en même temps que l'événement concret en question. Ceci est modélisé en signalant une erreur dans o u.t.. Ainsi, si un événement correspondant est trouvé en o ut, nous atteindrions un choix non-déterministe entre transiter vers `OK` ou encore vers `err`. Les deux transitions seront alors présentes dans le processus de composition. Comme nous le verrons, ce choix est résolu dans le propriété de μ-calcul par la recherche d'un chemin qui satisfait la simulation et en ignorant ainsi la transition d'erreur.

Time Observer Le `Control Observer` vérifie seulement si les deux événements correspondants peuvent se produire en même temps. Cependant, il ne dit rien sur le moment où un écoulement du temps s'est produit. Cela conduit à la définition d'un observateur supplémentaire `Time Observer` (Fig. 3.7) dans lequel deux aspects sont modélisés. Tout d'abord, à l'état initial `evt0`, seules les transitions qui sont tirables dans o u.t peuvent se produire. Ceci est fait en spécifiant un choix simultané entre un événement temporisé τ_0 contraint à $[0,0]$ d'un côté et tous les événements des systèmes abstraits et concrets d'un autre côté.

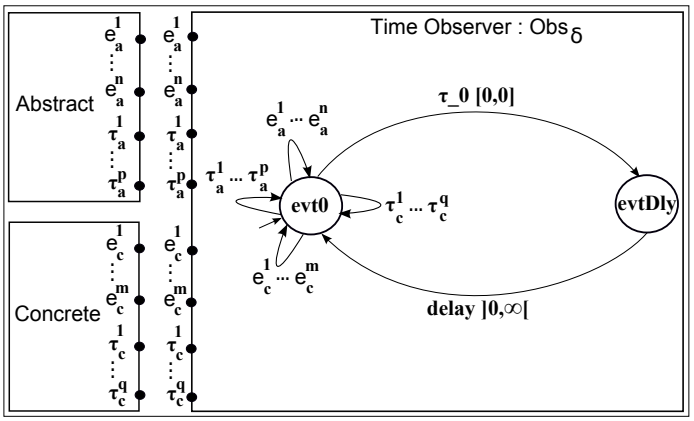


Figure 6 – *Time Observer*

Deuxièmement, elle rend visible l'écoulement implicite de temps. Dans l'état `Dlt`, à chaque passage du temps, un événement temporisé `delay` associé à la contrainte $]0,\infty[$ est signalé. Cet événement est utilisé plus tard par la propriété μ-calcul comme un marqueur d'écoulement de temps.

Critère de Vérification de la Simulation Faibe Temporisée

Le test de la vérification de la Simulation Faibe Temporisée consiste en une propriété μ-calcul sur le résultat de la composition du système abstrait, du système concret et des deux observateurs $(|||\ C) \| (Obs \underset{l_\tau A_r}{\|} Obs_\delta)$ où Obs est `Control Observer`, Obs_δ est `Time Observer` et $A_r = \{delay, \tau_0\}$.

La simulation des transitions de temps consiste à vérifier si chaque délai pris par le système concret peut également être fait par l'abstrait et conduire à un etat où la simulation reste verifiée. Le critère `Timed Faible Simulation` est défini comme suit:

$$\forall (q_a^0, q_c^0) \in Q_a^0 \times Q_c^0, (q_a^0, q_c^0, ok, evt0) \models \nu X. \overbrace{Obs\ in\ ok \wedge Obs_\delta\ in\ evt0}^{(1)} \wedge$$
$$\overbrace{\bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X) \wedge \bigwedge_j [\tau_c^j] X}^{(2) Weak\ Simulation} \wedge$$
$$\overbrace{(\mathbf{EF}_{\tau_a}\langle delay\rangle \top) \Rightarrow \mathbf{EF}_{\tau_a}(\langle delay\rangle \top \wedge [delay] X)}^{(3)}$$
$$\overbrace{\bigwedge_i [e_a^i]\langle e_c^i\rangle \top}^{Event\ \ Deadlock}$$
$$\overbrace{\bigwedge_{p \in \mathcal{P}} p_c \Leftrightarrow p_a}^{PropositionsRelation}$$

sachant que $(q_c, q_a) \models p_c$ si $p \in v_c(q_c)$ et $(q_c, q_a) \models p_a$ si $p \in v_a(q_a)$. Cette propriété caractérise un ensemble d'états du produit auquel l'état initial doit appartenir. Cet ensemble d'états est défini sur la composition des états de A, C et des deux observateurs. Nous commentons le critère:

- (1) désigne l'état dans lequel les événements concrets sont attendus.
- (2) est le critère de simulation faible non temporisée qui signifie que pour chaque cas événement e_c^i et pour chaque transition etiquettée par cet événement concret E_c^i, il existe un chemin d'un certain nombre d'événements locaux abstraites τ_a qui mène éventuellement à une transition marquée par l'événement abstrait e_a^i tel que la cible vérifie de manière récursive la propriété. En plus on demande qu'après chaque transition etiquettée avec un événement local τ_c^j la simulation est maintenue.
- (3) précise que si le temps peut s'écouler (cas de `delay`) dans le produit via une séquence d'événements locaux abstraits alors le temps peut s'écouler et pour tous les événements `delay` possibles la simulation est maintenue.
- (Event Deadlock) décrit la propriété de préservation du blocage. En fait, il décrit que chaque événement visible abstrait est suivi d'un événement concret correspondant.
- Propositions Relation : on dit que pour chaque proposition p, si p est satisfaite par les variables du système concret x_c^k, alors les variables du système abstrait x_a^l la satisfont aussi. Évidemment, les variables concrètes ou abstraites dépendent des systèmes qui sont comparés.

0.4 Méthodes Formelles pour la Vérification des Services Web

Les services Web sont des applications distribuées qui sont conçues pour réaliser une tâche spécifique de l'entreprise sur le web. Afin de réaliser les objectifs de l'entreprise, ces services Web doivent être composés de manière à interagir ensemble. Cette interaction se fait par échanges de messages via des interfaces publiques décrites dans langages basés sur XML tels que le Web Services Description Language WSDL [84].

L'orchestration est l'un des mécanismes qui addresse la composition du service. WS-BPEL (Web Services Business Process Execution Language) [18] ou tout simplement BPEL, est un langage de composition de service Webs qui suit le modèle de l'orchestration. Il définit les processus métier grâce à l'orchestration des différentes interactions partenaires.

Cependant, BPEL manque d'une sémantique formelle et est défini en langage naturel. D'une part, ses constructions peuvent souffrir d'ambiguïté, de l'incohérence et de l'incomplétude. Par exemple, comme indiqué par le comité *Web Services Business Process Execution Language Technical Committee*, de telles ambiguïtés et incomplétudes ont été détectés dans la première version de BPEL. Ces ambiguïtés ont été modifiées dans les versions ultérieures. D'autre part, la validité des exigences des utilisateurs sur du BPEL ne peut être vérifiée qu'après l'implementation effective du système. Cette faiblesse de BPEL peut être réparé par des méthodes formelles.

L'utilisation des méthodes formelles pour la vérification de services Web s'est avérée précieuse au sein de la dernière décennie. En fait, les méthodes formelles ont été la pierre angulaire et la base mathématique qui manquait dans le monde des services web.

Les méthodes formelles ont été utilisées afin d'élever le niveau de confiance des utilisateurs dans les applications web en particulier en raison des grandes quantités des mouvements monétaires qui transitent tous les jours via le web. La sémantique formelle pour les langages de composition de services, en particulier pour BPEL, ont été intensivement étudiés résultant en plusieurs ouvrages. Des approches fondées sur les réseaux de Petri, les algèbres de processus et les automates ont été proposées. Une transformation de descriptions informelles à des modèles formels a donc été introduite. En outre, puisque la sémantique formelle de ces formalismes est définie, les modèles formels peuvent alors être utilisés pour vérifier le bon fonctionnement des services Web. Cela se fait via la vérification des propriétés écrites en logique temporelle comme LTL et CTL.

Dans ce qui suit, nous présontons l'idée de la transformation de BPEL vers FIACRE.

0.5 Transformation de BPEL vers FIACRE

La modélisation décrite en FIACRE est basée sur la structure du processus BPEL. La partie WSDL définit les points de communication du processus BPEL. Ces connexions sont réalisées d'abord par la définition de Partnerlinks. Un Partnerlink définit le rôle que joue le processus (le cas échéant)

et le rôle que joue le service partenaire (le cas échéant) à l'échange particulier. Selon le rôle, un processus peut utiliser les PortTypes pour envoyer des appels/résultats ou recevoir des appels/résultats.

En conséquence, la partie statique de BPEL (WSDL) est modélisée dans FIACRE par des types globaux. Quant à la partie dynamique constituée de l'activité principale du processus BPEL et ses gestionnaires, ellecx est modélisée comme une composante racine en FIACRE contenant la composition du modèle FIACRE correspondant à l'activité primaire avec les modèles FIACRE correspondants aux gestionnaires (figure 7). Enfin, les points de communication d'un processus BPEL sont modélisés par deux ports FIACRE (`I` pour l'entrée (input) et `O` pour la sortie (output)).

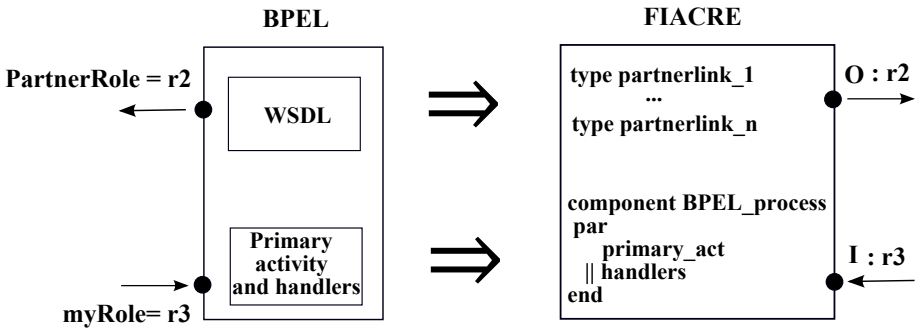


Figure 7 – *Structure de la Transformation*

0.5.1 Partie Statique : Modélisation du WSDL

L'interaction avec l'environnement est pris en charge par les Partnerlinks. En BPEL, chaque Partnerlink peut contenir deux rôles (`myrole` et `partnerRole`) typés avec des `Porttype`. Chacun des `Porttype` déclare plusieurs opérations servant à recevoir (Entrée) ou envoyer (sortie) des messages (figure 8).

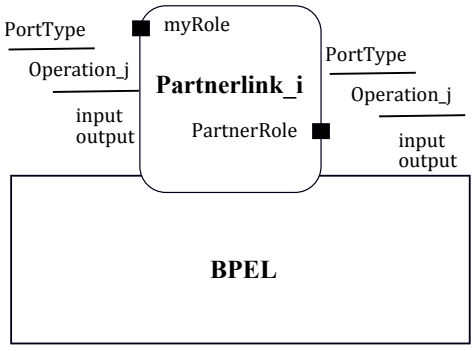


Figure 8 – *Connexions de BPEL*

Par conséquent, cette structure est modélisée dans FIACRE en créant deux types énumérés nommées `inputs` et `outputs` utilisés pour modéliser respectivement les entrées et les sorties de chaque opération.

Le type `inputs` (resp. `outputs`) sera l'union des types :

- des arguments `input` (resp. `output`) des opérations du `myRole` de chaque Partnerlink (1).
- des arguments `output` (resp. `input`) des opérations du `partnerRole` de chaque PartnerLink (2).

0.5.2 Aspects Comportementaux de BPEL en FIACRE

La transformation est guidé par les concepts de BPEL. Un processus BPEL est constitué de plusieurs activités BPEL réunies. En conséquence, chaque constructeur de BPEL se traduit séparément en un composant FIACRE.

1. Les activités de base seront converties en des `processus` de FIACRE.
2. Les activités structurées peuvent intégrer d'autres activités imbriquées. Par conséquent, ils seront chacun convertis en un `Component` de FIACRE.

Dans les deux cas, si une activité se traduit par un `processus` ou `component`, elle va partager une interface commune. Cette interface se compose de l'ensemble des ports FIACRE et des variables partagées que chaque modèle devrait avoir afin de permettre leur composition.

0.6 Vérification de BPEL

Les modèles transformés sont utilisées à des fins de vérification. Notre cadre de vérification prend en charge quatre types de propriétés.

1. Propriétés temporelles: ce sont les propriétés typiques rédigés en SE-LTL.
2. Propriétés liés aux données: au cas où les types de données BPEL sont finis, ils sont conservés dans la transformation BPEL / FIACRE. Il est alors possible de vérifier des propriétés qui traitent ces données.
3. Propriétés structurelles: ils expriment la "bonne construction" du code BPEL. Par exemple, nous sommes en mesure de vérifier qu'une activité `receive` est toujours de suivie d'une activité `reply`, ou que deux activités `receive` simultanées ne peuvent pas attendre pour la même opération. Cela se fait par une vérification statique que les excéptions résultants de ces situations ne sont pas levées.
4. Propriétés temporelles: elles peuvent être rédigées en MITL [16] qui est une variante temporisée de LTL et sont traitées ici en utilisant des observateurs temporisés et une propriété temporelle, codant ainsi des propriétés telles que les propriétés d'accessibilité.

0.7 Contributions et Résultas

- Contribution 1 : une technique automatique de vérification de la simulation faible temporisee (Timed Weak Simulation) entre des systèmes de transitions temporisés issus de systèmes FIACRE. La technique est une méthode basée sur l'observation, dans laquelle deux systèmes de transitions temporisés sont composées avec un observateur temporisé. Une propriété μ- calcul qui capte la simulation faible temporisée est ensuite vérifiée sur le résultat de la composition. Une caractéristique intéressante de la technique proposée est qu'elle repose uniquement sur un μ-calcul atemporel sans algorithme spécifique nécessaire pour analyser le résultat de la composition. Dans cette contribution, nous avons étudié les points suivants:

  1. Définition d'un cadre sémantique formel pour la langage FIACRE.
  2. Définition d'une relation de simulation qui implique des résultats intéressants concernant l'inclusion de trace, la préservation des propriétés et la compositionalité.
  3. La technique a été testée et validée en utilisant les outils FIACRE/TINA.

- Contribution 2: Application à la vérification de services Web:

  1. Une transformation de BPEL à FIACRE: la vérification est basée sur une transformation de BPEL vers FIACRE. Les avantages de cette transformation sont de deux ordres. Tout d'abord, nous proposons un cadre pour valider la bonne construction et la préservation par transformation de la sémantique d'un sous-ensemble des constructeurs BPEL. En outre, les constructions temporisées de BPEL sont prises en compte dans la transformation. Deuxièmement, la structure hiérarchique d'un processus BPEL est maintenud dans FIACRE. Les deux langages etant à base de composants, ceci rend la transformation modulaire.
  2. Cadre de vérification pour BPEL: On propose un cadre riche de vérification de BPEL, offrant plusieurs types de propriétés dont des propriétes temporisées.
  3. Une étude de cas montrant à la fois l'utilisation du test de simulation et la vérification de BPEL.

Contents

List of Figures

Part I

General Introduction

Chapter One

Introduction

Nowadays and with all the rapid technological progress that we are witnessing, it is safe for us to say that we are getting closer to be ruled by machines. Maybe not in the way the *terminator* movie shows it, but in the way of our reliance on the well functioning of automated systems in our daily life. These systems are getting complex by the day and are being utilized in numerous fields, such as telecommunications, avionics, automotive, aerospace, railways, medical, banking, web services and other related professions. Accordingly, a high degree of safety in these systems has become an indispensable prerequisite for their proper functioning.

In fact, software failures may result in profound deficit in the finance ranks or even in losses in the human life. According to COMPUTERWORLD, WASHINGTON in a report that dates back to 2/2002, "*software bugs are costing the U.S. economy an estimated 59.5 billion dollars each year, with more than half of the cost borne by end users and the remainder by developers and vendors*".

Several events in the past have shown the multitude of damage software errors can cause when they occur. In some cases, the damages have been described as catastrophic to say the least.

Let us remember the infamous 2003 Northeast blackout, where a massive power outage occurring on August 14, 2003, left vast regions of the Northeastern and Midwestern areas of the United States and Ontario Canada without power. Due to an error in programming, one system failed to take precedence over the other, leading to multiple systems trying to simultaneously access the same information at once. This havoc triggered an alarm malfunction whereby the software in question sent, in bulk, a busy signal to all systems. This costly technological blunder queued up unprocessed events and consequently slowed down the performance which caused the power outage. As a result, the losses were reportedly an estimated and painful seven to ten billion dollars.

Another notorious example is the crash of Ariane 5, a European satellite, which the European Space Agency spent ten years building with a total cost of a staggering eight billion dollars. The ultimate purpose of Ariane 5 was to give Europe a leadership role in the space business. Unfortunately, on June 4, 1996, Ariane 5 crashed only 36.7 seconds after being launched into the skies of French Guienna. The crash was the outcome of a 64-bit conversion floating point into a 16-bit integer value. Indeed, the number was too big resulting in an overflow error crashing both the pri-

mary and backup computers. The losses were 3650 days of hard work and labor as well as 500 million dollars worth of satellites and 8 billion dollars of production costs.

All these unfortunate events, which caused immense losses in time, labor, life, and dollars, are a proof that it is a responsibility to have such critical systems closely and sharply authenticated and validated before their definite implementation in the real world. In order to fulfill this imperative obligation, formal methods have been suggested.

1.1 Formal Methods

Formal methods are mathematically-based techniques, languages and tools for the specification and verification of computer-based software and hardware systems. Formal methods are used to secure a high-quality guarantee in the correctness of a system's specification. This increases confidence in its well-functioning through revealing the bugs that might otherwise go undetected.

1.1.1 System Specification

Formal methods may be used in the system specification phase when the requirements of a system are described. A specification is a formal model that represents the static and dynamic aspects of a system. While static aspects are represented by states, the dynamic aspects are represented by actions (transitions between the different states). Petri Nets [91], CSP [96], CCS [83] and B-Method [7] can be cited in this case as examples of model-based formal specification languages.

1.1.2 System Verification

Once a formal specification of the system is given, the specification may be validated by the proof of some properties. Several techniques for verification, which range from automatic to manual techniques exist. The system verification techniques fall into four general categories and these are theorem proving, model checking, abstract interpretation and software testing.

The theorem proving consists in proving the satisfaction of the properties on the systems by means of inference and induction rules. While some of these proofs are manual, others are either automated or interactive. There are several tools that offer the possibility to develop automated or interactive proofs. In an automated theorem proving tool, an automatic proof can be built given a description of the system, a set of logical axioms, and a set of inference rules. Examples of such tools are the Satisfiability Modulo Theories (SMT) [46] solvers (such as STP [51]) where the problem is to determine if a logical formula may be satisfied over a combination of theories expressed in classical first-order logic with equality. Unlike an automated theorem, an interactive theorem proving tool usually requires a human intervention. Examples of such tools are Coq [102], Isabelle [88], and B etc.

A second verification technique is abstract interpretation. Abstract interpretation [42] was formalized by both Patrick Cousot and Radhia

Cousot in the late 1970s. It is a method that is based on replacing an infinite domain of concrete values of a system by a number of finite abstract domains on which the initial operations (concrete) are redefined. Properties, such as interval bounds and null pointer access, are then verified on the abstract model.

Another technique is software testing. Software testing is the process of executing the real code of a program by specifying its parameters and verifying whether it behaves correctly. Testing allows a conformity check of the system compared to its definition and an examining of whether the software meets the requirements. The problem in this technique is the need to generate sufficient test cases in order to have a higher confidence in the product. However, since the number of test cases tends to be infinite, testing cannot be exhaustive (at the contrary of model checking : see below). Thus, it cannot prove that a product behaves correctly under all conditions but can rather show that it does under specific conditions.

We are going to discuss now the fourth verification technique, which is the technique adopted in this PhD.

Model checking

Model checking is an automated proof technique in which a property is checked against a system by means of an exhaustive search of all the possible states of a system execution. Formally, model checking refers to the following problem: given a model of a system M and a specification φ, we investigate whether the following equation is true, which is read as M satisfies φ or also as M is a model of φ.

$$M \models \varphi$$

The model is usually an abstraction of the system written in a formalism based on transition systems while the properties are logic properties written in LTL [79] or CTL [40]. The verification of a property on the system is done via model checking exploration algorithms. The first proposed algorithms were the ones of [40] and [94].

The advantages of model checking is that it is completely automatic and since it is an exhaustive verification it gives a high degree of confidence w.r.t the satisfied properties. Another advantage of model checking is that in the case of non-satisfaction of a property, a counter example is also given.

However, the main drawback of model checking is that its application is limited to only small models. As such, in case there are large models, model checking will suffer from combinatorial explosion. In order to trespass this drawback, several solutions mainly revolving around four techniques have been suggested. The first is abstraction techniques [41] which aim at reducing the size of the model. The second is partial order reduction techniques [89] which aim at reducing the size of the state-space to be searched by the model checking algorithm. The third is modular verification technique, where the idea is to decompose the system into smaller sub-components and to undergo the verification of each of the sub-component independently of the other. Finally, the fourth is the

refinement techniques which aim at building a model in an incremental manner.

A tool that implements the model checking technique is conventionally called a *model checker*. As examples of model checkers we can cite the Spin model checker [62] for transitions systems models, Tina [20] for Time Petri Nets, and UPPAAL [73] for timed automata.

1.2 Context and Problem

The context of this PhD work is the ANR project ITEmIS [2]. This project aims at easing the evolution from today's world of separate lightweight embedded applications and IT services to the future world of seamlessly integrated services. This transition qualifies and defines a new generation of service oriented architecture (SOA) that enables IT and Embedded Integrated Systems (ITEmIS systems). Thus, three main axis are considered for this purpose and these are the service infrastructure axis, the business axis, and the formal verification axis. The formal verification axis addresses end-to-end assurance of Qos and correctness of verification of the execution models. In this context, the needed work consisted at the ***verification of Business Execution Languages Processes*** (BPEL) via a transformation to a formal specification language FIACRE, a real time specification language. The results of this transformation (FIACRE models) could either be used to give a formal semantics for the BPEL constructs allowing the analysis of the BPEL constructs, or for verification purposes (model checking with logic properties). However, even though model checking may be considered for small BPEL models (FIACRE models after the transformation), the verification of larger models is not reasonable. The problem needed to be tackled differently, namely by considering the *incremental development* of FIACRE models, by means of progressive refinements. This has led to the study of the ***refinement*** of ***timed systems***. Let us start introducing the keywords of this PhD work.

1.2.1 Timed Systems

Even though classical models such as transition systems and Petri nets can express the behavior of a system, they cannot express its real-time constraints. However, time is a natural constraint that is explicitly present almost everywhere and in everything. Examples of timing constraints may be like execution delays, timeouts, system response and so on. In order to meet this requirement, timed systems were created in order to capture timing constraints that may control or alter the behavior of a system. Several models and logic were thus extended by timing notions (see next chapter).

Since dense time makes the state space infinite, its addition has surely made the comprehension and the theoretical aspects of timed models more complex. However, several results exist in the context of timed systems, most of which revolves around timed automata, one of the most studied models for real-time systems. The most interesting result is probably that reachability properties are decidable [15] and the decidability of TCTL (a timed variant of CTL) model-checking [12]. In contrast, language inclusion for timed automata is undecidable [15].

Consequently, the development of timed systems is surely harder, and its verification becomes more complex. The incremental development through time refinement is a technique that helps easing the development phase.

1.2.2 Refinement

As we have stated earlier in this chapter, the main drawback of model checking is the risk of combinatorial explosion. The incremental development has been suggested as a way to prevent the combinatorial explosion or at least to cope with it. Compositional verification and Refinement are two main techniques used for incremental development. Compositional verification allows the verification break up of a system into smaller tasks that involves the verification of its sub-components. Each of its small verifications would then allow inferring the global verification of the composition.

Refinement, which is the second approach, is what we consider in this PhD.

Stepwise-refinement allows building progressively correct specifications by making it more precise. At each stage of the building process, a new specification is derived from a former specification by verifying that each new one preserves the correctness w.r.t to the former one. In other words, the whole system is not modeled altogether, but rather, it is constructed gradually by a series of models, where each new model is supposed to be a refinement of the one preceding it.

1.2.3 Formal Methods for Web Services Verification

Web services are distributed applications which are designed to achieve a specific business task over the web. In order to carry out the business goals, these web services should be composed in such a way to interact together. This interaction is done by means of exchanging messages over public interfaces described in XML-based languages such as the Web Services Description Language WSDL[84]. Orchestration is one of the mechanisms that address service composition. WS-BPEL (Web Services Business Process Execution Language) [18] is a well known service composition language addressing orchestration. It defines business processes through the orchestration of different partner interactions.

However, BPEL lacks a formal semantics and is defined informally in natural language. At one hand, its constructs may suffer from ambiguity, inconsistency, and incompleteness. For example, as stated by the issues of the Web Services Business Process Execution Language Technical Committee such ambiguities and incompleteness were detected in the first version of BPEL. These ambiguities were amended in later versions. On the other hand, the validity of user requirements cannot be verified over BPEL processes before the actual implementation of the system. This BPEL weakness can be mended by formal methods.

The use of formal methods for web services verification has proven to be valuable within the last decade. Actually, formal methods were the mathematical cornerstone which the world of web services lacked.

They were used in order to elevate the degree of user confidence in web applications especially because of the large amounts of daily money transitions via the web. The formal semantics for web services composition languages, particularly for BPEL, were intensively studied resulting in several works. Approaches based on Petri Nets, process algebras, and automata were proposed. A transformation from informal descriptions to formal models was then introduced. Moreover, since the formal semantics of these formalisms is defined, the formal models may then be used to verify the well-functioning of the work-flow of web services. This is done via the verification of properties written in temporal logic such as LTL and CTL.

1.3 Contributions and Results

- Contribution 1 : an automatic technique for checking the timed weak simulation between timed transition systems based on models originating from FIACRE systems. The technique is an observation-based method in which two timed transition systems are composed with a timed observer. A μ-calculus property that captures the timed weak simulation is then verified on the result of the composition. An interesting feature of the proposed technique is that it only relies on an untimed μ-calculus model-checker with no specific algorithm needed to analyze the result of the composition. In this contribution, we study the following points :

  1. Definition of a basic formal semantic framework for the FIACRE language.
  2. Definition of timed weak simulation that supports interesting results concerning trace inclusion and the preservation of properties. Our simulation also enjoy the compositionality feature.
  3. The technique is tested and validated using the FIACRE/TINA tool-set.

- Contribution 2 : Application to the verification of web services :

  1. A Transformation from BPEL to FIACRE : the verification is based on a transformation of BPEL constructs to the specification language FIACRE. The advantages of this transformation are twofold. Firstly, we provide a framework for validating the correctness of a subset of constructs considered in the transformations. Moreover the BPEL timed constructs are taken into account in the transformation. Secondly, the hierarchical structure of a BPEL process is maintained in FIACRE because both languages are component-based languages making the transformation modular.
  2. Model Checking Verification Framework for BPEL : a rich verification framework is offered, taking into account several types of properties. Timed properties through the use of time observers, structural properties through annotations of the FIACRE code,

data-related properties (for finite types) and classical temporal properties such as deadlock freeness and so on .

The method adopted in this document is depicted in Fig 1.1. The first step is the transformation from BPEL to FIACRE. The result of this transformation is then abstracted and verified. The real implementation of the transformation's result is then tested for refinement against the abstract system.

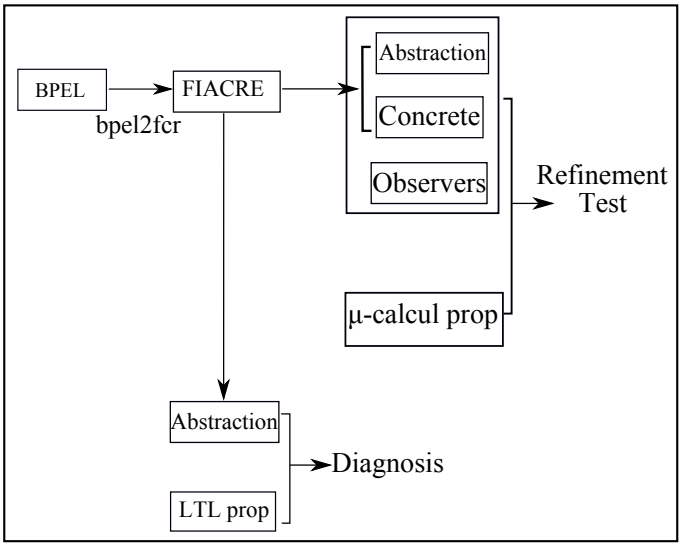


Figure 1.1 – *Overall Description*

1.4 Thesis Structure

The rest of this PhD document is organized in three parts.

Part II introduces the scientific context of our work.

In Chapter One, we introduce the timed systems. We start by introducing the linear temporal logic and its time variant. We also present in this thesis some real-time models such as Timed Automata and Time Game Automata considered in the thesis. We then proceed by presenting the syntax and semantics of the FIACRE language through different constructs.

We wrap up this chapter by presenting an alternative method that represents to represent the formal semantics of models and specification languages. As a matter of fact, we have identified a label-structure, based-construction method with which the formal semantics of our models could be eased. The complete work is given as an appendix : Appendix A.

Chapter Two is a presentation of the refinement and the simulation concepts. We begin by introducing the concepts in the untimed context. For this, we present the theory of refinement in both the state and action based semantics. We later discuss about some of the existing studies in the timed context.

Chapter Three includes our contribution in the topic of timed simulation definition and verification. In this chapter, we start by formalizing the context of the study. For this, we establish a definition to what we call Constrained Time Transition System. We then give our definition of

timed weak simulation and show that it guarantees traces inclusion and thus preserving linear time logic. We also show that our parallel operator is monotonic w.r.t the simulation. Later in the chapter, we suggest a technique for the verification of our timed simulation and give the correctness and completeness proof of the technique.

Part III consists in the integration of the timed simulation in the verification chain of BPEL processes.

Chapter One consists of an introduction to service oriented architecture. For this, we define the web services and their composition. This will lead us to the BPEL, a web service composition language. After giving a brief introduction to BPEL and its constructs, we finish this chapter by discussing the relation of web services to formal methods. We then present some of the existing approaches in this context.

In Chapter Two, we present the transformation from BPEL to FIACRE. We show a one to one transformation for most of the BPEL constructs. In this transformation, the static part of BPEL (WSDL) is transformed into FIACRE data types while the dynamic part of BPEL (activities) is transformed to FIACRE components. We also show how some of the FIACRE patterns may be validated w.r.t their BPEL semantics.

Chapter Three presents the verification framework of BPEL. Here we illustrate how the timed properties are treated via the use of observers, whether at the FIACRE or BPEL level. We also show how the structural properties are verified via explicit annotations of the FIACRE code.

To finalize this part and to illustrate our approach, we present in Chapter Four a case study that shows a transformation from BPEL to FIACRE, through the use of our refinement technique. The use case is then model checked using the Tina model checker.

In Part IV, we formulate a conclusion to the thesis and discuss several perspectives of these works.

Part II

Theoritical Aspects

Chapter One

Timed Systems

1.1 Introduction

In order to respond to the need of real time constraints in the study of hardware and software systems, several timed models and timed logic were created. These were mostly extensions of existing formalisms used in the untimed context. In these extensions, a quantitative elapse of time has been integrated, either in the form of discrete time (for example, see [59]) or in the form of dense time (for example, see [14]). The dense nature of time is closer to what we have in real life and represents an uncountable set of values in an interval. This means that at any instant, it can be divided to smaller units. This is not the case in a discrete time view since on the contrary, a discrete time has a countable domain, like integers for example. For this, the notion of dense time is more coherent for the representation of real time systems. Here, we only consider formalisms where the time is dense.

In this chapter we present some of the logics and their extensions that are used to specify timed properties. We also present the timed models used to specify the timing behavior of systems.

1.2 Temporal Logics

Temporal logic is a term to denote a category of logic that allows to represent temporal information within a logical framework. This means that the propositions are qualified in terms of time. For example, if the proposition is, "it will rain", then statements like, "it will ***always*** rain" or "it will ***eventually*** rain" could be expressed in temporal logic.

Consequently, temporal logics have proven valuable in formal verification of reactive systems, since they allow to express intuitive properties such the mutual exclusion, the response properties, and so on. There are two main temporal logics which are the branching temporal logic and the linear time logic. In the linear view, a model is a sequence-like structure such as that at each moment there exists only one successor in time, whereas in the branching view a model has a tree-like structure, where future may split into multiple branches.

1.2.1 Linear Temporal Logic

Linear Temporal Logic (or Linear-time Temporal Logic) is a modal temporal logic with modalities referring to time. In LTL, one can encode formula about q given future. For example, a condition will eventually be true, a condition will be true until another fact becomes true, and so on.

We present in this section different variants of linear temporal logic [80]. Such logic will play a major role in this PhD. The first role pertains to the motivation of the simulation definition, since we are interested in choosing a definition that preserves such linear time logic properties. Linear time logic will also be used to express the properties and the requirements that need to be verified by a system.

Linear Temporal Logic (LTL)

We start by describing a state based LTL where atomic properties are observers on the system state.

Syntax The syntax is defined by means of the following grammar :

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \,\mathbf{U}\, \varphi_2$$

Where $p \in P = p_1, \cdots, p_n$ is the set of propositions.
The LTL formulas are defined inductively in the following manner :

1. $\top$ is a formula.
2. A proposition $p \in P$ is a formula.
3. If φ is a formula, then $\neg\varphi$ is also a formula.
4. If φ_1 and φ_2 are formulas, then $(\varphi_1 \vee \varphi_2)$ is also a formula.
5. If φ is a formula then $\bigcirc\varphi$ is also a formula.
6. If φ_1 and φ_2 are formulas, then $\varphi_1 \mathbf{U} \varphi_2$ is also a formula.

From these we can derive other operators such as :

1. $\bot$ is equal to $\neg\top$.
2. $\varphi_1 \wedge \varphi_2$ is equal to $\neg(\neg\varphi_1 \vee \neg\varphi_2)$.
3. $\varphi_1 \Rightarrow \varphi_2$ is equal to $\neg\varphi_1 \vee \varphi_2$.
4. $\Diamond\varphi$ is equal to $\top \mathbf{U} \varphi$.
5. $\Box\varphi$ is equal to $\neg(\top \mathbf{U} \neg\varphi)$.
6. $\varphi_1 \,\mathbf{R}\, \varphi_2$ is equal to $\neg(\neg\varphi_1 \mathbf{U} \neg\varphi_2)$.
7. Weak Until : $\varphi_1 \,\mathbf{W}\, \varphi_2$ is equal to $(\varphi_1 \mathbf{U} \varphi_2) \vee \Box\varphi_1$.

LTL Semantics Given a set of states S, $Tr = S^\omega$ the set of infinite sequences of elements of S, and $v : S \to 2^P$, a valuation function that associates to each state its set of satisfied propositions, let $\pi = s_0, s_1, s_2 \dots$ a sequence of states and φ an LTL formula. We define φ *is true on* π denoted as $\pi \models \varphi$, recursively on φ. For all $i = 0, 1, ...$ we denote by π_i the sequence of states s_i, s_{i+1}, s_{i+2}....

1. $\pi \models \top$.
2. $\pi \models p$ if $p \in v(\pi_0)$.
3. $\pi \models \varphi_1 \vee \varphi_2$ if $\pi \models \varphi_1$ or $\pi \models \varphi_2$.
4. $\pi \models \neg\varphi$ if $\pi \not\models \varphi$ (or$\neg\pi \models \varphi$).
5. $\pi \models \bigcirc\varphi$ if $\pi_1 \models \varphi$.
6. $\pi \models \varphi_1 \mathbf{U} \varphi_2$ if $\exists k \mid \pi_k \models \varphi_2$ and $\forall k' < k \ , \pi_{k'} \models \varphi_1$.

State-Event Linear Temporal Logic (SE-LTL)

We have just seen the semantics of the state-based LTL. Here, we extend this logic with event-based semantics leading to the definition of the SE-LTL logic [37]. SE-LTL is an extension of LTL. The only syntactic difference between LTL and SE-LTL is that the latter can have both state propositions and event propositions allowing to refer to both states and events when constructing specifications. An important distinction to be made between the state-based traces and the event based traces is that in state properties, it is possible to have multiple propositions satisfied at the same state, while event properties are exclusive. Formally, the SE-LTL logic can be defined in the following manner :

SE-LTL Syntax The syntax of SE-LTL is defined by means of the following grammar :

$$\varphi := \top \mid e \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \, \mathbf{U} \, \varphi_2$$

Where $e \in E = e_1, \cdots, e_n$ is the set of events and $p \in P = p_1, \cdots, p_n$ a set of propositions. The SE-LTL formulas are defined inductively in the same way as in the Event LTL syntax.

SE-LTL Semantics Given a set of states S, $Tr = (S \times E)^\omega$ the set of alternating infinite sequence of states and events, and $v : S \to 2^P$ a valuation function that associates to each state its set of satisfied propositions, let $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \cdots \in Tr$ a sequence of state/events in the set of traces Tr and φ a SE-LTL formula. For all $i = 0, 1, ...$ we denote by π_i the sequence of states/events $s_i \xrightarrow{e_i} s_{i+1} \xrightarrow{e_{i+1}} s_{i+2}$.... In addition to the semantics of LTL, we have the following :

1. $\pi \models e$ if $\pi = s_0 \xrightarrow{e} ....$

Time Domain

Before introducing timed logic, we introduce the timed domain describing the operations available over the time data type. We define a time domain [64] as a commutative monoid $(\Delta, 0, +)$ with 0 as a neutral element. We also assume :

1. $+$ is left-cancelative : $t + u = t + v \Rightarrow u = v$.
2. $+$ is anti-symmetric : $t + u = 0 \Rightarrow t = u = 0$.
3. Δ is ordered : $u \leq v \triangleq \exists x, u + x = v$.
4. $\leq$ is total : $u \leq v \vee v \leq u$.

Thus $(\mathbf{N}, 0, +)$ and $(\mathbf{R}^+, 0, +)$ are respectively a discrete and dense time domains.

Metric Interval Temporal Logic (MITL)

We have seen that in LTL and SE-LTL, an execution of a system is modeled by a sequence of states or events. As noted, in this model, the precise timing of occurrence of events or propositions is not specified. To address this, many logics were extended by timing notions. This has lead to the creation of the Metric Temporal Logic MTL [72]. MTL extends LTL by adding interval of times to the temporal operators.

Unfortunately, it has been proven that the satisfiability and model checking problems for MTL are undecidable [55]. Thus, some restrictions on MTL were considered. It turns out that the problem of decidability arises from what is called punctuality or punctual intervals [17, 56] which is the ability to specify that a particular event is always followed exactly one time unit later by another one: $\Box(p \rightarrow \Diamond_1 q)$. But this is a little bit out of the scope of this document. Here, we just say, that a restriction to MTL which only considers non punctual intervals has been defined. This has lead to the Metric Interval Temporal Logic MITL [16]. As LTL, we start by describing a state-based version of MITL.

Syntax The syntax is defined by means of the following grammar :

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \, \mathbf{U}_I \, \varphi_2$$

Where $p \in P = p_1, \cdots, p_n$ the set of propositions and I is a non punctual interval in Δ with closed or opened bounds. The MITL formulas are defined inductively in the following manner :

1. $\top$ is a formula.
2. A proposition $p \in P$ is a formula.
3. If φ_1 and φ_2 are formulas, then $\varphi_1 \vee \varphi_2$ is also a formula.
4. If φ is a formula, then $\neg\varphi$ is also a formula.
5. If φ_1 and φ_2 are formulas, then $\varphi_1 \mathbf{U}_I \varphi_2$ is also a formula.

As in the case of LTL, we can derive other operators. Here, we just show the operators specific to the MITL logic.

1. $\Diamond_I \varphi$ is equal to $\top \mathbf{U}_I \varphi$.
2. $\Box_I \varphi$ is equal to $\neg(\top \mathbf{U}_I \neg\varphi)$.
3. $\varphi_1 \mathbf{R}_I \varphi_2$ is equal to $\neg(\neg\varphi_1 \mathbf{U}_I \neg\varphi_2)$.

Intuitively, the formula $\Box_I \varphi$ means that φ is always satisfied after any delay of interval I. The formula $\Diamond_I \varphi$ means that φ will be satisfied after some delay of interval I. Finally, $\varphi_1 \mathbf{U}_I \varphi_2$ means that φ_1 is always satisfied until a certain point in time t where φ_2 becomes satisfied. This point of time t needs to happen in the interval I.

MITL Semantics Given a set of properties $\mathcal{P}$, the semantics of MITL formulas are defined over timed state sequences [16]. A timed state sequence is a sequence $\sigma = (P_0, I_0) \rightarrow (P_1, I_1) \rightarrow ...$ where $P_i \subseteq \mathcal{P}$ and I_i are closed intervals such that:

1. I_0 is left-closed and its left bound is equal to 0,
2. For all $i > 0$, the right bound of the interval I_i is equal to the left bound of the interval I_{i+1},
3. Each instant $t \in \mathbb{R}^+$ is an element of a unique interval I_i of the timed state sequence.

This means that each interval of the timed state sequence represents the states when some propositions are satisfied. Given a time $t \in I_i$, we denote by σ^t the suffix of a state sequence from the instant t : $\sigma^t = (P_i, I_i - t) \rightarrow (P_{i+1}, I_{i+1} - t) \rightarrow (P_{i+2}, I_{i+2} - t) \ldots$ and $t \in I_i$. We denote as well by σ_i the i^{th} couple (P_i, I_i) of the timed state sequence σ. Formally, for $disc((s, I))$ a function that returns the discrete part s of an element of a timed state sequence and $v : S \rightarrow 2^P$ a valuation function that associates to each state its set of satisfied propositions, the MITL operators are defined over timed state sequences σ in the following manner :

1. $\sigma \models \top$.
2. $\sigma \models p$ if $p \in v(disc(\sigma_0))$.
3. $\sigma \models \varphi_1 \vee \varphi_2$ if $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$.
4. $\sigma \models \neg\varphi$ if $\sigma \not\models \varphi$ (or$\neg\sigma \models \varphi$).
5. $\sigma \models \varphi_1 \mathbf{U}_I \varphi_2 : \exists t \in I$, $\sigma^t \models \varphi_2$ and $\forall t' \in [0, t[$, $\sigma^{t'} \models \varphi_1$.

Stave/Event Metric Interval Temporal Logic (SE-MITL)

Just like the case of LTL, we define a version of MITL based on its event semantics.

Syntax The syntax is defined by means of the following grammar :

$$\varphi := \top \mid e \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2$$

Where $e \in E = e_1, \cdots, e_n$ the set of events, $p \in P = p_1, \cdots, p_n$ the set of propositions and I is an interval in $\mathbb{R}^+$ with closed or opened bounds. The SE-MITL formulas are then defined inductively in the same way as in the MITL case.

SE-MITL Semantics Just as before, the semantics of SE-MITL formulas are defined over timed state event sequences. A timed event state sequence is a sequence $\sigma = (P_0, I_0) \xrightarrow{e_0} (P_1, I_1) \xrightarrow{e_1} ....$ In addition to the semantics of MITL, we have the following :

1. $\sigma \models e$ if $\sigma = (P_0, I_0) \xrightarrow{e} ....$

1.2.2 Branching Temporal Logic

There exists several branching temporal logic. The most known maybe Computation tree logic CTL [40] or CTL variants and μ-calculus [32]. CTL is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any of which might be an actual path that is realized. In this document, we will just discuss the μ-calculus logic which we will use in the contribution part of this PhD.

μ-Calculus

Modal/Propositional mu-calculus (L_μ) [32] is a branching time temporal logic used extensively in different fields of theoretical computer science, in systems verification and model checking. This is mostly due to its high expressive power, thereby, many branching time logic can be translated into this logic. It can be viewed as a generalization of CTL thanks to the fix-point operators μ and ν allowing to define respectively branching liveness and safety properties.

Syntax of μ-Calculus

Let Var be a set of variable names, typically denoted by Z, Y, ...; let Prop be a set of atomic propositions, typically denoted by P, Q,...; and let L be a set of labels, typically denoted by a, b, ... The set of L_μ formulas (w.r.t. Var, Prop,L) is defined as follows:

$$\varphi ::= P \mid Z \mid \varphi_1 \wedge \varphi_2 \mid [a]\varphi \mid \neg\varphi \mid \nu Z.\varphi$$

From these we can derive other dual operators such as :

1. $\varphi_1 \vee \varphi_2$ means $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$.
2. $\langle a \rangle \varphi$ means $\neg[a]\neg\varphi$.
3. $\mu Z.\varphi(Z)$ means $\neg\nu Z.\neg\varphi(\neg Z)$

The meaning of the formula $[a]\varphi$ is that φ holds after all a-actions. Its dual $\langle a\rangle\varphi$ means that it is possible to do an a-action to a state where φ holds. Recursion operators are used for recursive formula $\mu Z.\varphi(Z)$ and $\nu Z.\varphi(Z)$ where ν (resp. μ) is the greatest (resp. least) fixed point operator.

Semantics of μ-Calculus

The models for the μ-calculus are defined over a structure $\mathfrak{S}$ of the form $\langle S, L, T, v\rangle$ where $\langle S, L, T\rangle$ is a labeled transition system (introduced in Section 1.3) and $v : Prop \to 2^S$ is a valuation function that maps each atomic proposition P $\in$ Prop to sets of states where P holds. Given a structure $\mathfrak{S}$ and a function $\mathfrak{V} : Var \to 2^S$ that maps the variables to sets of states in the transition system, the set $\|\varphi\|^{\mathfrak{S}}_{\mathfrak{V}}$ of states satisfying a formula φ is defined as follows :

- $\|P\|^{\mathfrak{S}}_{\mathfrak{V}}$ = V(P).
- $\|X\|^{\mathfrak{S}}_{\mathfrak{V}} = \mathfrak{V}(X)$.
- $\|\neg\varphi\|^{\mathfrak{S}}_{\mathfrak{V}}$ = S - $\|\varphi\|^{\mathfrak{S}}_{\mathfrak{V}}$.
- $\|\varphi_1 \wedge \varphi_2\|^{\mathfrak{S}}_{\mathfrak{V}} = \|\varphi_1\|^{\mathfrak{S}}_{\mathfrak{V}} \cap \|\varphi_2\|^{\mathfrak{S}}_{\mathfrak{V}}$.
- $\|[a]\varphi\|^{\mathfrak{S}}_{\mathfrak{V}} = \{s \mid \forall t,\, s \xrightarrow{a} t \Rightarrow t \in \|\varphi\|^{\mathfrak{S}}_{\mathfrak{V}}\}$.
- $\|\nu X.\varphi\|^{\mathfrak{S}}_{\mathfrak{V}} = \bigcup\{Q \in 2^S \mid Q \subseteq \|\varphi\|^{\mathfrak{S}}_{\mathfrak{V}[X\mapsto Q]}\}$ where $\mathfrak{V}[X \mapsto Q]$ is the valuation which maps X to Q and otherwise agrees with $\mathfrak{V}$.

Notations We define the notations **EF** (exists finally) and **AG** which will be helpful in the coming sections. These operators (existential and universal quantifiers) are CTL-like operators in a state-event based variant.

$$\mathbf{EF}_L P = \mu Z.P \vee \bigvee_{l\in L} \langle l\rangle Z \qquad (\textbf{Liveness})$$

This is read as there exists (expressed by $\langle l\rangle$) a finite path labeled by elements of L after which a state is reached where P holds. Since we have used a μ this also means that P will eventually hold. We also define :

$$\mathbf{AG}_L P = \neg\mathbf{EF}_L\neg P = \nu Z.P \wedge \bigwedge_{l\in L} [l]Z \qquad (\textbf{Safety})$$

This means that on all the paths that are labeled by $l \in L$, P holds on all of its states. If L is the full set of labels, we get the usual definition of the CTL EF and AG operators.

1.3 Timed Systems Models

In this section, we study briefly some timed system models. These are Timed Automata, Timed Game Automata, and Timed transition systems, followed by the FIACRE language, a component-based real time system modeling language. For each of these models, we will give the syntax and the semantics.

1.3.1 Timed Transition Systems

We start by giving the formal definitions of a transition system and its timed variant, a timed transition system, which will be used to give the semantics of the timed systems models given here. Formally, a labeled transition system is :

Labeled Transition System LTS Given a set of labels L and a set of atomic propositions P, a Labeled Transition System can be described as $\langle Q, Q^0, \rightarrow \rangle$ where :

- Q is a set of states,
- $Q^0 \subseteq Q$ is a set of initial states,
- $\rightarrow \subseteq Q \times L \times Q$ is a set of transitions.

We denote by τ the silent action and define $L_\tau = L \cup \{\tau\}$. We write $q \xrightarrow{l} q'$ for $(q, l, q') \in \rightarrow$ and denote $q \xrightarrow{l}$ iff there exists a state q' such that $q \xrightarrow{l} q'$. If $l \neq \tau$ then $q_i \stackrel{l}{\Rightarrow} q_j$ is a shorthand for $q_i \xrightarrow{\tau}{}^* \xrightarrow{l} q_j$ otherwise it is a shorthand $q_i \xrightarrow{\tau}{}^* q_j$. We define as well the direct successors of a state by the following $Post(q, \alpha) = \{q' \in Q \mid q \xrightarrow{\alpha} q'\}$ and $Post(q) = \bigcup_{\alpha \in L} Post(q, \alpha)$. A state q of a transition system is called terminal if $Post(q) = \emptyset$.

Now we extend the labeled transition system to the timed context by adding time labels to the transitions.

Timed Transition System TTS Let Δ be a time domain, a timed transition system TTS is a transition system where the transition relation $\rightarrow$ is defined as $\subseteq Q \times (L \cup \Delta) \times Q$. This means that a TTS is an LTS having two kinds of transitions. Control transitions represented by action labels and time transitions represented by continuous delay. Furthermore, we require standard axioms for $\rightarrow$ as defined in [44]:

1. time determinism : for all $q, q', q'' \in Q$ and for all $\delta \in \Delta$ whenever $q \xrightarrow{\delta} q'$ and $q \xrightarrow{\delta} q''$ then $q' = q''$.
2. time reflexivity (zero-delay) : for all $q, q' \in Q$, $q \xrightarrow{0} q' \equiv q = q'$.
3. time additivity : for all $q, q', q'' \in Q$ and for all $\delta_1, \delta_2 \in \Delta$ if $q \xrightarrow{\delta_1} q'$ and $q' \xrightarrow{\delta_2} q''$ then $q \xrightarrow{\delta_1 + \delta_2} q''$.
4. time continuity : for all $q, q' \in Q$ and for all $\delta \in \Delta$, if $q \xrightarrow{\delta} q'$ then for every $\delta_1, \delta_2 \in \Delta$ such that $\delta = \delta_1 + \delta_2$, there exists q'' such that $q \xrightarrow{\delta_1} q'' \xrightarrow{\delta_2} q'$.

1.3.2 Timed Automata

A timed automaton [15] is a formalism for the modeling of reactive and real-time systems. It is a finite automaton having a finite set of real-valued clocks, that is, variables with values $\in \mathbb{R}^+$. In a timed automaton, time is always synchronous, meaning that all the clocks increase at the same rate.

Moreover, the values of these clocks may be tested during a run of a timed automaton using a clock constraint language which forms the guard of the transition. The clocks may also be reset. Timed automata have gained a lot of popularity in recent years.This popularity for timed automata is mainly because -thanks to restrictions on the constraints language of the guards- its reachability problem is decidable [15]. This is why, lots of verification techniques, such as checking safety and liveness properties, simulation and bisimulation of timed automata have been a subject of study during the last decade. Two main tools that are based on timed automata have also been developed. These are the model checkers UPPAAL [73] and KRONOS [104].

Formal Definition

Formally, a timed automaton is a tuple A = $(\Sigma, Q, q_0, X, I, T)$ that consists of the following components :

1. Σ is a finite alphabet of actions.
2. Q is a finite set of states.
3. $q^0 \in Q$ is an initial state.
4. X is a finite set of clocks.
5. $I : Q \rightarrow C(X)$ is a mapping function that assigns to each state a clock constraint in X called the location invariant.
6. $T \subseteq S \times \Sigma \times C(X) \times 2^X \times Q$ is a set of transitions. Every element $e = \langle q, a, \varphi, \lambda, q' \rangle \in T$ corresponds to a transition from the state q to the state q', with a label a, a guard φ, and a reset set $\lambda \subseteq X$.

Clocks and Constraints

A clock x is a variable with a value $\in \mathbb{R}^+$. For a set X of clocks, $C(X)$ is the set of clock constraints that is defined by the following grammar :

$$\phi_1, \phi_2 \in C(X) ::= true \mid x < c \mid x \leq c \mid x > c \mid x \geq c \mid \phi_1 \wedge \phi_2$$

where $x \in X$, c is a constant $\in \mathbb{Q}$.

Semantics

The semantics of a Timed automaton A is a timed transition system with an infinite set of states in the form of pairs (q, k) where $q \in Q$ is a state of A and k is a clock interpretation function over X that assigns a real value to each clock such that $k \models I(q)$. The transitions of the timed transition system can be either discrete transitions or time transitions. Consider a state (q, k). Given a transition $t = (q, a, g, r, q')$ of A and $d \in \mathbb{R}^+$:

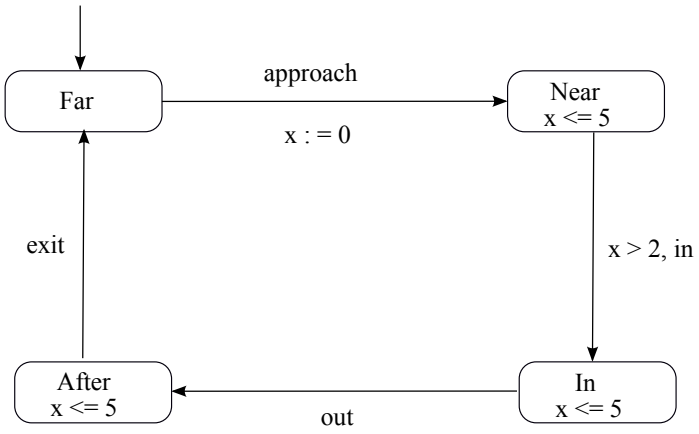
- $(q, k) \xrightarrow{a} (q', k')$ is a *Discrete Transition* where $(k' = [r := 0]k)$ if $k \models g$ and $k' \models I(q')$.
- $(q, k) \xrightarrow{d} (q, k + d)$ is a *Time transition* if $k + d \models I(q)$ and for all $0 \leq e \leq d$, $k + e$ satisfies the constraint $I(q)$.

Example

This is a classic example for reactive systems which is the Railroad Crossing [11]. It consists of three processes, a train, a gate and a controller. The idea is that whenever the train arrives, the controller needs to guarantee that the gate is open. In addition, the gate needs to be closed when the train leaves. Here, we just show the modeling of the train. The real time properties which are needed to be satisfied are the following :

1. The train signals its approach for at least 2 units of time after entering the gate (expressed through the approach action).
2. No more than 5 units of time should elapse between the approach and the exit (expressed by the clock associated to the exit action).

Figure 1.1 – *Train Modeling in Timed Automata*



The clock is initialized to 0 when the approach signal is received. The train cannot enter the gate, if since its approach, the total elapsed time is less than 2 units of time (satisfies the first property) which means it cannot go too fast (minimal bound). Moreover, the time constraint associated to the "In" state specifies that the total elapsed time since the approach must be less that 5 units of time. The train leaves the "In" state respecting its invariant. In the "After" state, the total elapsed time must always be less than 5 units of time which means the train must go fast enough (maximal bound). The *exit* event occurs at most 5 u.o.t after the approach signal, thus satisfying the second property.

1.3.3 Timed Game Automata

Here, we talk about the timed game automata [78] or the Larsen's and al variant of timed input output automata [44]. Actually, these two have similar definitions. For the case of the input/output automata, its actions are split between input and output actions. Whereas in the case of timed game automata, its actions are split between controllable and uncontrollable actions. Intuitively, the output actions correspond to controllable ones while the input actions, since they are sent by the environment, correspond to non-controllable ones. Even though these two formalisms are statically different, they share a lot of common notions, especially the alternating

simulation definition and the algorithm of its verification. In [44] the theory of timed input output automata is presented where the simulation and different composition operations over timed input output automata are defined. However, the alternating simulation algorithm is based on a timed game presented in [36] and implemented later in the UPAAL-TIGA tool [22].

Timed Game Automata Definition

A Timed Game Automaton (TGA) G is a timed automaton with its set of transitions T partitioned into controllable (T_c) and uncontrollable (T_u) transitions. Formally, it is defined as a timed automaton A = $(\Sigma, Q, q^0, X, I, T)$ where $\Sigma = Act_c \oplus Act_u$ is a finite set of actions, partitioned into controllable and uncontrollable respectively.

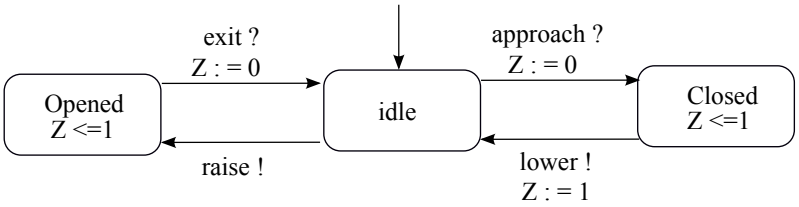
Semantics of a TGA

Similarly, the semantics of timed automaton is a timed transition system, by applying the same rules, the semantics of TGA is a transition system where its set of actions are partitioned into controllable and uncontrollable.

Example

We give the modeling in terms of a timed input output automata of the controller of the Railroad Crossing. The controller is initially at the state `idle`. Upon receiving the train's `approach` signal, it transmits the `lower` signal asking to lower the gate. When the train's `exit` signal is received, the controller transmits the `raise` signal asking to raise the gate. The response time of the controller to the `approach` signal is 1 minute, and to the signal `exit` is at most 1 minute. These constraints are expressed using the clock z.

Figure 1.2 – *Controller Modeling in Timed Input Output Automata*



1.3.4 Other Timed Models

Let us start by reviewing some notions in their original untimed versions.

CSP

CSP (Communicating Sequential Process) [96] is a process algebra for describing concurrent systems that was developed by Tony Hoare in an article published in 1978. Combining a synchronization mechanism based

on "appointments" with a simple and concise syntax, CSP allows an implementation of traditional paradigms of concurrency. For this purpose, it offers operators such as prefixing $x \rightarrow P$ which is a process that synchronizes on the event x and then behaves as the process P, the recursion ($P = x \rightarrow P$) which is a process that can perform an indefinite number of x, deterministic $P \Box Q$ and non deterministic choice $P \sqcap Q$ between processes, deadlock process (STOP) and parallel composition ($P \parallel Q$), and so on.

Example: Here is an example of a coffee machine that offers the choice between a big and a small cup of coffee. The machine accepts a 2 euro coin and returns depending on the choice of the customer a big cup of coffee or a small one with a 1 euro change. It accepts as well a 1 euro coin and returns a small coffee or even two successive 1 euro coins and returns a big coffee. However, a foolish customer insists at having a big cup of coffee independently of the coins he enters. The composition of these processes results in a satisfied customer upon the customer's payment of 2 euros. Otherwise the composition will block.

$$\begin{aligned} COFFEE \;=\; & (\; in2p \;\rightarrow (large \;\rightarrow\; COFFEE \\ & \qquad \Box\; small \;\rightarrow\; out1p \;\rightarrow\; COFFEE) \\ & \Box\; in1p \;\rightarrow\; (small \;\rightarrow\; COFFEE \\ & \qquad \Box\; in1p \;\rightarrow\; (large \;\rightarrow\; COFFEE \\ & \qquad\qquad \Box\; in1p \;\rightarrow\; STOP))) \end{aligned}$$

$$\begin{aligned} FOOLCUST \;=\; & (in2p \;\rightarrow\; large \;\rightarrow\; FOOLCUST \\ & \Box\; in1p \;\rightarrow\; large \;\rightarrow\; FOOLCUST) \end{aligned}$$

$$SYSTEM = (FOOLCUST \;||\; COFFEE)$$

Petri Nets

A Petri Net [91] is a directed graph that contain places and transitions that may by connected by directed arcs. Transitions represent actions while places represent states. The arcs run from a place to a transition or vice versa. Places may contain tokens, and a transition can be fired if there is at least one token at each of its input places. In this case, the tokens may move to other places. Formally, a Petri Net is a tuple (S, T, F, M_0) where:

- P is a set of places.
- T is a set of transitions.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.
- $M_0 : S \rightarrow \mathbb{N}$ a function called *initial marking* that associates to each place a set of tokens.

The marking of a Petri Net describes the number (possibly zero) of tokens inside the places. A place is then empty or marked. If the place represents a logical condition, (for ex : machine off), the presence of token means that this condition is true while its absence means that the condition

is false. If the place represents a resource (for ex : stock), it can contain multiple tokens.

Example This example of Fig 1.3 shows how a waiter in a restaurant may serve the customers. A customer can only be served if a waiter is free. As an initial marking, we suppose that the waiter is free and that the two customers have arrived to the restaurants.

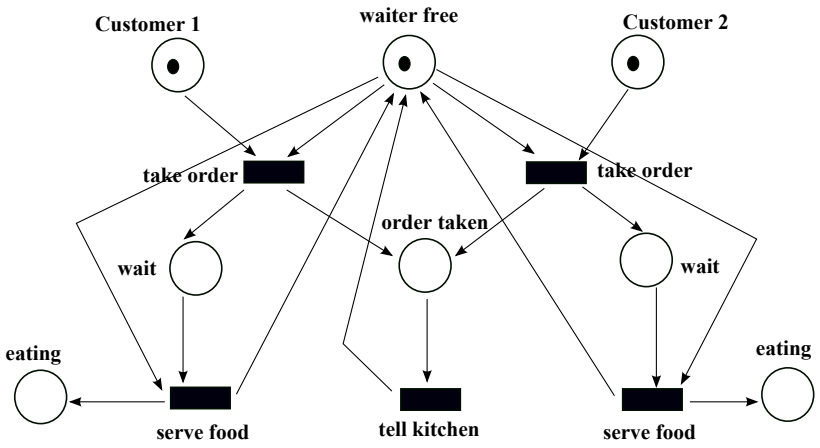


Figure 1.3 – *Restaurant in Petri Net*

Because of concurrency, there exists several possible executions of this example. Here, we only discuss two of them

- Scenario 1: the waiter takes the order from customer 1 (places : `wait` and `order taken`) before telling the kitchen (place : `waiter free`). The waiter then serves customer 1 (places : `eating` and `waiter free`). He then takes order from customer 2 and serves customer 2 in the same way.
- Scenario 2: the waiter takes the order from customer 1 (places : `wait` and `order taken`) before telling the kitchen (place : `waiter free`). The waiter takes order from customer 2 (places : `wait` of customer 2 and `order taken`), tells the kitchen (place : `waiter free`) and serves customer 2 (places : `eating` of customer 2 and `waiter free`). Afterwards, he serves customer 1 since a token is still available in the `wait` place of customer 1.

Timing Extensions

Adding Time to Transition Systems Timed transitions systems [45] [1] are an extension of transitions systems where real time constraints are added by associating minimal and maximal time delays with the transitions. The timing constraints means that an enabled transition should be fired after

[1] The name is an overload of the timed transition systems that we have seen earlier in this chapter. Actually, the timed transition system of [45] is a higher level model than the one we have seen.

its minimal bound and before its maximal bound unless it gets disabled. In Timed Transition Systems, we suppose that all transitions happen instantaneously while the timing constraints restrict the time in which the transitions can be made.

Adding Time Constraints to Petri Nets As other timed models we can cite the extensions that were made to existing models used in the untimed context. For example, different Petri Nets time extensions were suggested [24] such as adding a duration to a place [71], a duration of a transition (Timed Petri Nets [95]) or also a delay to a transition (Time Petri Nets [81] where a time interval is associated with each transition specifying the minimal and the maximal time elapsing after the enablement of the transition. Time Petri Nets are supported by the Tina tool (Section 1.3.6) and are used to define the semantic model of FIACRE (Section 1.3.5). Formally, a time Petri Net is a couple $TP = (R, \delta)$ where $R = (S, T, F, M_0)$ is a petri net and δ is a function that associates a time interval to each transition. In the example of Fig 1.4, given a function f that returns the absolute firing time of a transition then if f(T0) = 12 and f(T1) = 4 then f(T2) = 5, 6 or 7 since the tokens of P2 are created by either one of the transitions T0 and T1. We note here that since the maximal bound of T1 (6) is less than the minimal bound of T0 (10) then T0 is never fired even though a token is present in P0.

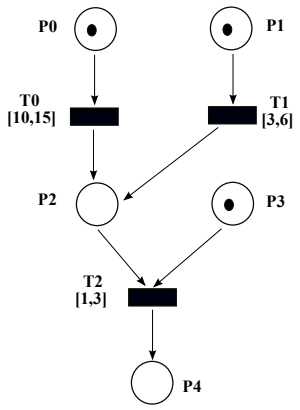


Figure 1.4 – *Time Petri Net*

Adding Time to CSP Process algebras such as CSP were also extended by timing notions which lead to Timed CSP [5]. In Timed CSP, the operators were extended by time notions and new operators were added to describe timeouts, explicit waiting (**wait t**), timed interrupt and so on.

1.3.5 The FIACRE Language

FIACRE [29], an acronym for Format Intermédiaire pour les Architectures de Composants Répartis Embarqués (Intermediate Form for Architectures of Embedded Distributed Components) is a formal intermediate language dedicated to the modeling of both the behavioral and timing aspects of systems. It is used for formal verification and simulation purposes. Basically, FIACRE is a process algebra where hierarchical components commu-

nicate either through synchronous messages by means of ports or through shared memory by means of variables. These ports and variables define the interface of a component. A FIACRE program is a set of components declarations (specifications) together with a "main" component. A specification is either a Process which are the leaves of the FIACRE program or Component which are its nodes.

- **Process** : describes a sequential behavior using symbolic transition systems based on [58]. Accordingly, a process is defined by a set of states and transitions. Each transition has a guard and a non deterministic action that may synchronize on a timed port and may update local or shared variables.
- **Component** : describes the parallel composition of sub-components. Moreover, the components may introduce variables and ports shared by its sub-components. Real time constraints and priorities can be attached to ports. The real time constraint attached to a port `p` is a time interval `I`. This means that once the event `p` is enabled, the execution should wait at least the minimum bound of `I` and at most its maximum bound.

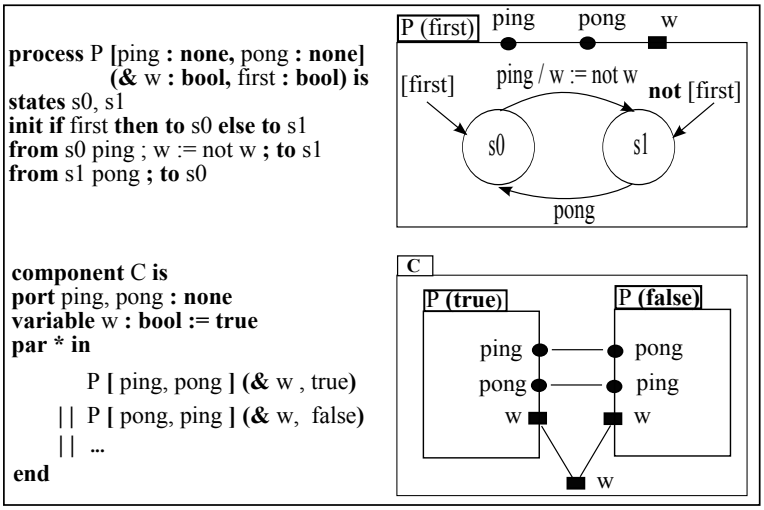


Figure 1.5 – *Graphical notations*

In the example given in Fig 1.5, we show the textual syntax and the graphical syntax where processes are illustrated as automatons and components as communicating boxes through ports (•) or shared variables (■). We will use both notations in the rest of this document. In Fig. 1.5, two process instances of `P` are composed together by synchronizing on the ports `ping, pong` and by sharing the variable `w`. The Boolean variable `first` allows to specify the instance of the process `P` that initiates the game. Once this is done, the game proceeds by doing an infinite number of `ping` and `pong` events. The `ping` of an instance is the `pong` of the other.

Adding Timing Constraints If we want to create a ping pong game that is able to finish, we may do this by adding timing constraints that alter the

course of the game. For example, in Fig 1.6, if either one of the ping and pong events is not made in 1 u.o.t the game may (or must, depending if priorities are added) end via the transition to the state `finish`.

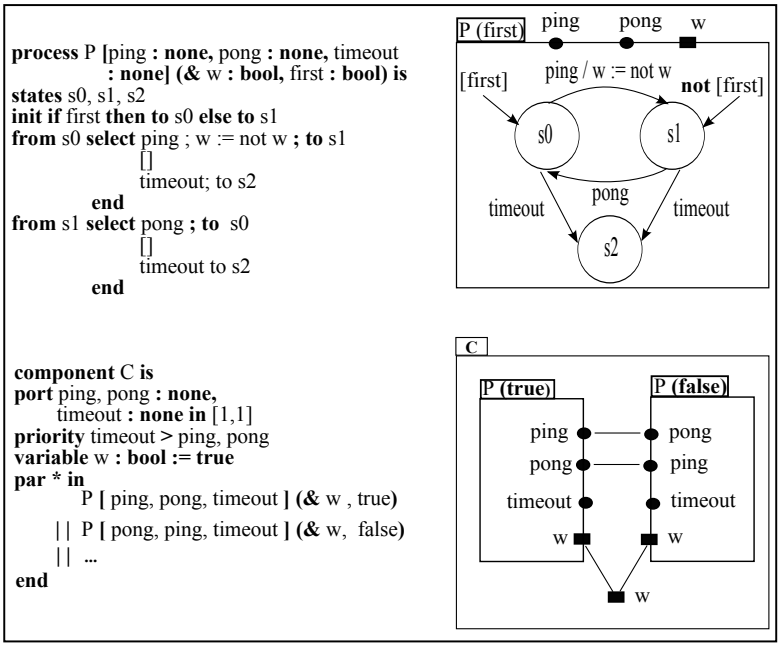


Figure 1.6 – *Timing Constraints in FIACRE*

1.3.6 TINA

TINA (TIme petri Net Analyzer) [20] is a toolbox developed at LAAS/CNRS which allows the edition and analysis of Petri Nets and Time Petri Nets extended with data variables and priorities. The essential tools of TINA that are used in this PhD are :

1. frac : FIACRE to tina compiler which compiles FIACRE specifications into high level time petri nets, the input language of most TINA tools.

2. tina : a state space abstraction tool that generates a finite automaton. As in most model checking tools, abstractions techniques are used to help prevent state-explosion. However, these abstractions need to preserve specific properties of the original specifications. To cope with state explosion, TINA offers various abstract state space constructions that preserve specific classes of properties of the state spaces of nets, like absence of deadlocks, linear time temporal properties, and bisimilarity. For timed systems, TINA also provides various abstractions based on state classes preserving reachability properties, linear properties, or branching properties.

3. selt : a linear time logic model checking tool. Selt is a tool that implements a state/event variant of linear time logic LTL. This means, that both state-based properties, such as the value of data-variables or event-based properties, such as the occurrence of events, may be

checked using this logic. Moreover, real time properties expressed in timed variants of CTL or LTL, are checked using the standard technique of observers, encoding such properties into reachability properties.

4. muse : a modal mu-calculus model checker. It is used to model check the finite automata by modal mu-calculus formula. The muse tool must be used to model check abstractions of the system preserving branching properties (using the -A attribute) since mu-calculus is a branching logic. Moreover, the priorities in this case cannot be used.

1.4 Alternative Definitions : Use of Label Structures

We are going to talk about an alternative work in the context of the study of formal semantics of specification languages. As a matter of fact, we have identified a construction method with which the formal semantics of our models could be eased. This method is based on abstracting and encapsulating the labels and the compositional behavior of systems in a label structure.

The label structure is equipped with a composition operator which specifies the specific composition laws of each language and would further serve as a parameter of the behavioral framework. Such structure could be instantiated differently depending on the choice of the target language. Thanks to the separation between the composition laws and the behavioral framework, the latter, which is based on LTS, offers a unique partially synchronous LTS composition which is reused to define syntactic composition of different extensions of LTS.

We show how the LTS and TTS notions are built following this method. For the interested readers, we give in Appendix A a more complete work in which we show how timed automata can be built by label structures. We start with the definition of a label structure :

Definition 1.1 (Label Structure) *A label structure is a tuple $\langle L, L_E, \bowtie \rangle$ where L is a set of labels, $L_E \subseteq L$ is a set of errors and $(\bowtie : L \times L \nrightarrow L)$ denotes a partial binary composition operator over L.*

In this structure we differentiate between a normal label and an error label. The addition of a separate error label at this level may seem unnatural for those who are interested in model checking tooling. But note that in this case, it is purposely done for semantic definition goals. The partiality of the function comes from the fact that some composition may be blocked since the composition operator $\bowtie$ only describes the synchronous aspects of the composition. As for the asynchronous aspects, they are described later in the systems composition. The creation of an error label stems from the fact that in certain languages, namely FIACRE, a distinction is made between execution errors (blocking) and compilation errors. For instance, reading two different values of the same variable leads to a blocking transition, while concurrent reading and writing and concurrent writing of the same variable is an error.

We show an example of a label structure which is used later when defining a timed model :

Time Label Structure. For Δ a time domain (non-negative real numbers, naturals ...) equipped with a binary associative operator $+$ and a neutral element 0, the time structure TS is given in the following where a synchronization only takes place when the same value is present at the two sides of the composition :

$$TS = \langle \Delta, \emptyset, (\delta_1, \delta_2) \mapsto \delta_1 \textbf{ if } \delta_1 = \delta_2 \rangle$$

Composition of Label Structures The label structures can also be composed to create a new label structure. In practice, this is very helpful and very useful, especially, when building progressively the models (for example, by refinement). We define the product and the sum of two label structures. The product operation builds new labels as couples of the composed labels. This is particularly used when composing synchronization and memory access labels. Unlike the product operation, the labels of the sum operation are defined over the disjoint union of the composed labels. This is especially used when composing synchronization and time labels to specify that only one of the events may occur but not both simultaneously.

Product of Label Structures Given two label structures $\langle L, L_E, \bowtie \rangle$ and $\langle L', L'_E, \bowtie' \rangle$, their product ranges over the set $P = (L \cup \{\epsilon\}) \times (L' \cup \{\epsilon'\}) \backslash \{(\epsilon, \epsilon')\}$ where ϵ (resp. ϵ') is a new element of L (resp. L') supposed to be neutral for the $\bowtie$ operator of its respective label structure. For l_1, $l_2 \in L$ and l'_1, $l'_2 \in L'$, the composition of $(l_1, l'_1), (l_2, l'_2)$ is defined only if the composition of l_1 and l_2 and the composition l'_1 and l'_2 are also both defined.

$$\langle L, L_E, \bowtie \rangle \otimes \langle L', L'_E, \bowtie' \rangle = \langle P, L_E \times (L' \cup \{\epsilon'\}) \cup (L \cup \{\epsilon\}) \times L'_E,$$

$$(l_1, l'_1), (l_2, l'_2) \mapsto (l_1 \bowtie l_2, l'_1 \bowtie' l'_2) \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie) \wedge (l'_1, l'_2) \in \textbf{dom}(\bowtie') \ \rangle$$

Sum of Label Structures Given $\langle L, L_E, \bowtie \rangle$ and $\langle L', L'_E, \bowtie' \rangle$, their sum ranges over the disjoint union of $L^\bullet \uplus {}^\bullet L'$ where $L^\bullet = \{l^\bullet \mid l \in L\}$ and ${}^\bullet L' = \{{}^\bullet l \mid l \in L'\}$. Formally, the sum is defined as:

$$\langle L, L_E, \bowtie \rangle \oplus \langle L', L'_E, \bowtie' \rangle =$$

$$\langle L \uplus L', L_E \uplus L'_E, \begin{pmatrix} l_1{}^\bullet, l_2{}^\bullet \mapsto (l_1 \bowtie l_2)^\bullet \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie) \\ {}^\bullet l_1, {}^\bullet l_2 \mapsto {}^\bullet(l_1 \bowtie' l_2) \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie') \end{pmatrix} \rangle$$

The Sum operator would be used later to build the labels of a time labeled transition system where its labels can be an event or a time value.

Labeled Transition Systems (LTS)

Now, we reformulate the definition of the labeled transition system by taking into account our label structure. This leads into an LTS defined over a label structure and that inherits its compositional behavior out of the composition operator of the label structure.

Definition 1.2 (Labeled Transition System LTS) *Given a label structure* $LS = \langle L, L_E, \bowtie \rangle$*, a labeled transition systems* $\mathcal{L}$ *over LS -denoted as* $\mathcal{L}_{LS}$*- is defined as* $\langle Q, Q^0 \subseteq Q, T, \alpha : T \to Q, \beta : T \to Q, \lambda : T \to L \rangle$ *where :*

- Q, Q^0, T *denote respectively the sets of states, initial states, transitions.*
- α, β *and* λ *denote functions that return respectively, the source, the target and the label of a transition.*

Timed Transition System The construction could be extended into the timing model TTS, by making use of our defined time label structure.

Definition 1.3 (Timed Transition Systems TTS) *Given a label structure* $LS = \langle L, L_E, \bowtie \rangle$*, a Timed Transition System TTS over LS is an LTS over* $LS \oplus TS$*.*

We finish this section by discussing about the composition of the different behavioral systems. In fact, since the composition is made at the label structure level, the composition of different extensions of labeled transition systems (for example, the time transition system) would reuse the composition defined at the labeled transition systems. This means that the behavioral composition itself is intact. However, it is parameterized by a label structure containing the composition laws specific to each extension of labeled transition systems. For more details, see Appendix A.

1.5 Conclusion

We have seen in this chapter some of the logic and timed models that are used in real time modeling and verification. Most of these definitions are time extensions of general definitions in the untimed context. We have also defined the FIACRE language which will be the used language in this document. In the next chapter, we cover the simulation relations between transitions systems.

Chapter Two

Refinement and Simulation

2.1 Introduction

The verification of real-time systems plays a major role in the conception of highly trusted systems. Yet, the more complex the system is, the more complex -in terms of space and time- and the less tractable its verification tends to be. New techniques have been suggested in order to minimize, in terms of space and time, the verification cost. Refinement is one of the most interesting concepts.

Refinement is a technique that allows the incremental development of systems. Indeed, it allows to model progressively -at different levels of abstractions- the behavior of the system. When a system C is said to be a refinement of an abstract system A, then all of the properties satisfied by the abstract system are also satisfied by the concrete level. More generally, we say that if C is a refinement of A then each time the system A is used, the system C can be used instead. In model checking, this is very useful since the errors could be found earlier in the modeling of the system, without any concern about its implementation.

Several relations and sufficient conditions were suggested for refinement checking. This chapter consists in an introduction to the simulation relations in both the untimed and the timed context. These relations are sufficient conditions for the refinement of systems.

2.2 Simulation in the Untimed Context

The refinement in the untimed context has been studied intensively in the literature. This has lead to several relations, equivalences and preorders defined on labeled transition systems that differ in their semantics and the properties they preserve.

Generally, we can find two approaches followed for the definition of these relations. These approaches are either state-based or action-based. Classically, when the main interest is mostly model checking these relations are usually defined based on the observable state properties of the transition system whereas its actions are ignored. Conversely, when the interest is process algebras for example, these same relations are found but focus on the actions of the systems and ignore the states. [52] provides

an overview and comparison of existing behavioral equivalences in the state-based context.

Whether in the action-based context or in the state-based context, a wide range of behavioral equivalences and relations has been developed to compare two systems. Here, we only emphasize on the simulation relations.

Simulation relations are often called implementation relations which are used commonly for verification purposes. By implementation relations, it is meant that they are used to show that a concrete system implements an abstract system. Even though simulations are not equivalence relations, they enjoy an important feature which is their transitivity thus making them applicable in incremental developments via intermediate refinements. When a concrete model is proven to be simulated by an abstract one -on which the properties are verified-, the latter can be replaced by the concrete model.

In what follows, the simulations are all defined on labeled transitions systems LTS which we have introduced in the Timed Systems Chapter. Throughout this chapter, the transition systems under consideration may have terminal states. However when talking about the temporal logic that are preserved by these relations, the transitions systems are assumed to be without terminal states. We explain the reason of this assumption in the following sections. Concerning the temporal logic preservation, here we are interested in the preservation of LTL, the logic adopted in this PhD. When a relation preserves other logic than LTL we will just cite it but we will not go beyond this since this is out of the scope of this PhD.

2.2.1 Action-Based Simulation

We start by introducing action-based definitions of the simulation relations. In the action-based view [61], a system is studied based on the sequence of actions it makes. There exists two variants of definitions for the traces model depending on whether the silent actions are taken into consideration or not. In the strong semantics, the system is studied based only on its observable actions. However, apart of the observable actions that a system can induce, it may also have an invisible behavior that is based on silent actions or moves. A silent action, (usually denoted as τ) means that the system may transit to another state without being noticed by the environment. In the context of process algebra this may be the result of a Hiding or a Renaming construct used on actions [61, 83]. This has lead to the weak semantics.

Strong Simulation

Given a set of labels L, two LTS Systems $A = (Q_A, Q_A^0, \rightarrow_A)$ and $C = (Q_C, Q_C^0, \rightarrow_C)$, a relation $R \subseteq Q_C \times Q_A$ is a strong simulation relation [82] iff $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for every $(q_C, q_A) \in R$ and for every action $l \in L$, if $q_C \xrightarrow{l} q'_C$ then there exists q'_A such that $q_A \xrightarrow{l} q'_A$ and $(q'_C, q'_A) \in R$.

We write $C \precsim A$, if there exists a strong simulation relation R between C and A. Here we note that a relation $R \subseteq S_C \times S_A$ is a strong bisimulation

relation iff both R and R^{-1} are strong simulations. We write $C \sim A$, if there exists a strong bisimulation relation R between C and A.

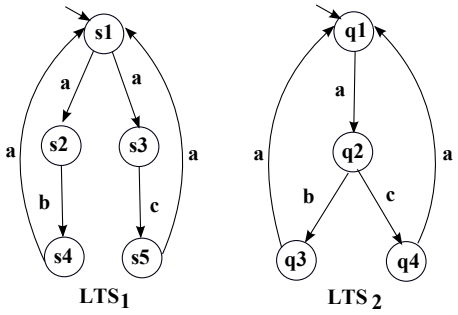


Figure 2.1 – *Strong Simulation*

In the example of Fig 2.1, $LTS_1 \precsim LTS_2$ through the relation $R = \{(s_1, q_1), (s_2, q_2), (s_3, q_2), (s_4, q_3), (s_5, q_4)\}$. However, $LTS_2 \not\precsim LTS_1$ since there is no state in LTS_1 that can simulate the state q_2 in LTS_2.

Preserved Logic The strong simulation preserves the linear temporal logic (LTL) if the systems have no terminal states. Actually, the strong simulation preserves a fragment of the CTL* logic [40] (Universal/Existential fragment of CTL*) which is more expressive than LTL. Here we do not discuss about the CTL* since it is out of the scope of this document.

Now let us explain by means of an example the reason behind the assumption of transitions systems without terminal states.

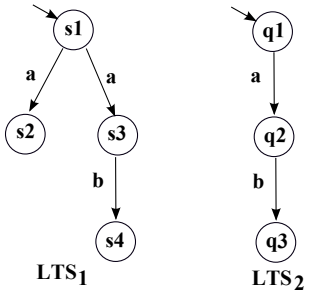


Figure 2.2 – *Terminal State*

In the example 2.2, $LTS_2 \precsim LTS_1$ and all the properties satisfied by LTS_1 are preserved in LTS_2. However, while also having $LTS_1 \precsim LTS_2$, $q_2 \models \bigcirc b$ but $s_2 \not\models \bigcirc b$. This is because s_2 is a terminal state in LTS_1 but q_2 is not in LTS_2. In this case, the strong simulation only preserves a fragment of LTL which corresponds to the safety properties. The same explanation will hold for all the relations we see later in this chapter. As an alternative to the assumption of not having terminal states in the LTS, an additional condition may be added to the definition the simulation [23]. This assumption implies that any deadlock in the concrete system corresponds to a deadlock in the abstract system which means that new deadlocks are forbidden. For every $(Q_C, Q_A) \in R$ this is formulated as the following : $q_C \not\rightarrow \Rightarrow q_A \not\rightarrow$. Note when adding this clause we would no longer have $LTS_1 \precsim LTS_2$ since $s_2 \not\rightarrow \not\Rightarrow q_2 \not\rightarrow$.

Weak Simulation

Given a set of labels L_τ, two LTS Systems $A = (Q_A, Q_A^0, \rightarrow_A)$ and $C = (Q_C, Q_C^0, \rightarrow_C)$, a relation $R \subseteq Q_C \times Q_A$ is a weak simulation relation [83] iff $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for every $(q_C, q_A) \in R$ and for every action $l \in L_\tau$, if $q_C \xrightarrow{l} q'_C$ then there exists q'_A such that $q_A \stackrel{l}{\Rightarrow} q'_A$ and $(q'_C, q'_A) \in R$.

We write $C \precsim_w A$, if there exists a weak simulation relation R between C and A.

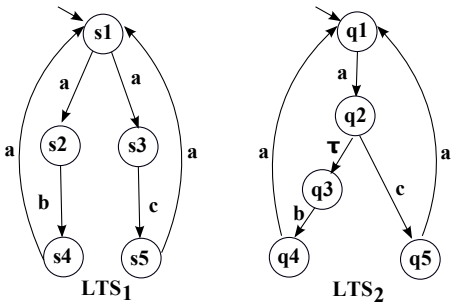


Figure 2.3 – *Weak Simulation*

In the example of Fig 2.3, $LTS_1 \precsim_w LTS_2$ through the relation $R = \{(s_1, q_1), (s_2, q_2), (s_2, q_3), (s_3, q_2), (s_4, q_4), (s_5, q_5)\}$. However, $LTS_2 \not\precsim_w LTS_1$.

Preserved Logic The weak simulation only preserves a fragment of LTL [53]. Actually, the liveness properties and, more precisely the eventual operator is not preserved by the weak simulation. This results from the presence of divergent paths (infinite path with only silent actions). To better understand this, we again take an example :

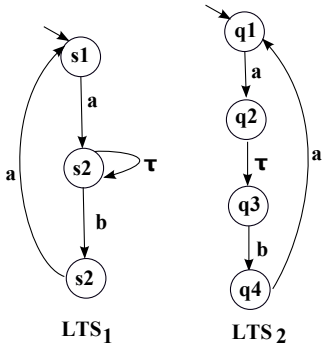


Figure 2.4 – *Divergent Path*

In the example 2.4, $LTS_1 \precsim_w LTS_2$. However, $LTS_2 \models \Diamond b$ but $LTS_1 \not\models \Diamond b$. This is because LTS_1 may be stuck forever in s_2 by doing silent actions, without ever leaving, and thus b may never be satisfied. This is why, another hypothesis is added so that the weak simulation may preserve LTL. This hypothesis forbids an infinite path composed only with a cycle of silent transitions in the concrete system [23]. With the addition of this hypothesis LTS_1 would no longer weakly simulate LTS_2 in the example 2.4. We will revisit this when discussing the weak simulation in

the timed context and when giving our simulation definition in the CTTS timed simulation chapter of the PhD (Chapter three of Part I).

2.2.2 State-Based Simulation

The same behavioral definitions of the previous section may be given by following a state-based approach [21]. In the state-based approach, the behavioral relations on transition systems are given by referring to the state labels. This means, the definitions are based on the propositions that are satisfied in a state.

Simulation Order

Given a set of atomic propositions P, two transitions systems $\langle Q_C, Q_C^0, \rightarrow_C \rangle$ and $\langle Q_A, Q_A^0, \rightarrow_A \rangle$ with $v_C : Q_C \rightarrow 2^P$ and $v_A : Q_A \rightarrow 2^P$ two labeling functions specifying the propositions that hold in a state, a simulation relation [21] is a binary relation $R \subseteq Q_C \times Q_A$ such that $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for all $(q_C, q_A) \in R$:

- $v_C(q_C) = v_A(q_A)$.
- if $q'_C \in Post(q_C)$ then $\exists q'_A \in Post(q_A)$ such that $(q'_C, q'_A) \in R$.

We write $C \preceq A$, if there exists a simulation relation R between C and A.

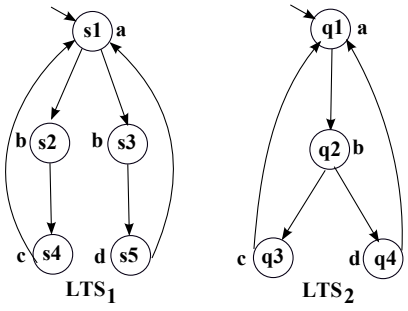


Figure 2.5 – *Simulation*

In the example of Fig 2.5, $LTS_1 \preceq LTS_2$ through the relation $R = \{(s_1, q_1), (s_2, q_2), (s_3, q_3), (s_3, q_4), (s_4, q_5)\}$.

Preserved Logic Just like its action-based variant, the simulation order preserves the linear temporal logic (LTL). The same discussion applies concerning the terminal states.

Stutter-based Simulations

Just like in the case of the action-based approach, the state-based relations may be weakened by admitting the fact that invisible actions may happen in the transition system. The state-based relations that we have seen so far are strong variants of these relations. That is, they require that each step at one level is matched by exactly one step at the second level. For instance, if s_C and s_A are in relation, then weakening this condition means that we admit that a step from s_C may be matched by a sequence of steps

which are invisible from s_A. The steps (except the last state change) in such sequences should not change the truth value of the atomic propositions that hold in s_A. Such state changes are called stutter steps.

Stutter Simulation Given a set of atomic propositions P, two transitions systems $\langle Q_C, Q_C^0, \rightarrow_C \rangle$ and $\langle Q_A, Q_A^0, \rightarrow_A \rangle$ with $v_C : Q_C \rightarrow 2^P$ and $v_A : Q_A \rightarrow 2^P$ two labeling functions specifying the propositions that hold in a state, a stutter simulation relation [34],[77] is a binary relation $R \subseteq Q_C \times Q_A$ such that $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for all $(q_C, q_A) \in R$:

- $v_C(q_C) = v_A(q_A)$.
- if $q'_C \in Post(q_C)$ and $(q'_C, q_A) \notin R$, then $\exists q_A q_A^1 \cdots q_A^n q'_A$ with $n \geq 0$ and $(q_C, q_A^i) \in R,\ i = 1 \cdots n \wedge (q'_C, q'_A) \in R$. [1]

The second condition means that if q_C and q_A are related with R then for every outgoing transition from q_C to q'_C, it must be matched by a path fragment from q_A to q'_A such that q'_C and q'_A are equivalent and all intermediate states in the path fragment are equivalent to q_C.

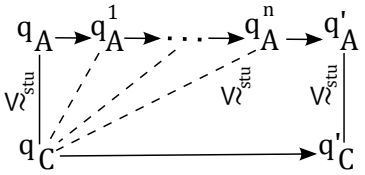


Figure 2.6 – *Second Condition of the Stutter Simulation*

We write $C \preceq_{stu} A$, if there exists a simulation relation R between C and A.

Observational Simulation Observational equivalence (simulation) was originally defined in the action-based semantics by [83]. Here, we talk about its state-based reformulation which can be seen as a variant of the stutter bisimulation. Unlike the stutter bisimulation, in the observational equivalence if q_C and q_A are related with a relation R and we need to simulate the transition from q_C to q'_C, then it is not required that the intermediate path fragment from q_A to q'_A to be equivalent to q_A.

Preserved Logic Just like the weak simulation, both of the stutter simulation and the observational simulation do not preserve LTL [21]. As before, the reason behind this is the divergent stutter paths. To understand the problem of divergent paths in the state based semantics, we reformulate the example of Fig 2.4 in the state-based semantics which results of Fig 2.7. In the example of Fig 2.7, $LTS_1 \preceq_{stu} LTS_2$. However $LTS_2 \models \Diamond b$ while $LTS_1 \not\models \Diamond b$. For this fact, the divergence sensitive stutter simulation is defined. The divergence-sensitive (DS) stutter simulation is a variant of stutter simulation which restricts the stutter simulation by allowing the states that belong to the same equivalence class (same propositions satisfied) to be related only if they both exhibit divergent paths or none of

[1] We note here that $q_A q_A^1 \cdots q_A^n q'_A$ is a finite path fragment

them has a divergent path. Thus, the simulation between LTS_1 and LTS_2 is rejected based on the DS-Stutter simulation since s_1 exhibits a divergent path while q_1 does not. The DS stutter simulation preserves LTL and a fragment of CTL*.

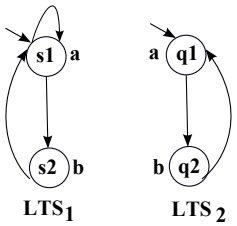


Figure 2.7 – *Stutter Simulation and DS Stutter Simulation*

2.2.3 Simulation and μ-calculus

μ-calculus was used as means for simulation verification in the Mec 5 model checker [54, 28]. This model checker handles specifications written in Altarica [19] and embeds the μ-calculus logic as a support for properties verifications on Altarica models. The authors propose several relations checked in Mec 5 which are the strong simulation, weak simulation and the bisimulation. In order to say that an Altarica model refines another, one of these relations need to be established on the free product of the LTSs corresponding to the two Altarica models. We add also that the proposed relations are event-based, meaning that starting from two states c and c' that are in a relation, we check if for each possible event e of the abstract system (enabled from state c), a corresponding event of the concrete system can be found e' enabled from the state t' such that the sink states of c and c' are also in a relation. Concerning the flow variables in Altarica, they are supposed to be equal between the abstract and concrete system. An example of a Mec 5 formula *Sim* for the strong simulation equation is given below :

$$Sim(c,c') = eqC(c,c')\&([e][t](transA(c,e,t) \Rightarrow \langle e'\rangle\langle t'\rangle(transA'(c',e',t')\&Sim(t,t'))));$$

where $eqC(s,s')$ specifies the set of state pairs such that both states associate the same values to the flow variables, $transA$ and $transA'$ define respectively transitions for nodes A and A'. Once all the pairs of states in strong simulation are computed, the strong simulation between A and A' is verified if their initial states verify the simulation.

Other μ-calculus properties were also used as a mechanism to capture and to check simulation relations. We can cite the work of [70] where a triple-nested μ-calculus formula is given to check the fair simulation [57].

2.3 Simulation in the Timed Context

Timed simulation relations have been studied for different timed formalisms ranging from timed transition systems, to timed automata [15,

77], timed input output automata TIOA [68, 43] and timed modal specifications [30]. However, a less tackled aspect of this research is the automatic verification of timed simulations and especially timed weak simulations. In the following sections, we take a look on how the simulation relations are extended to the timed context. We also present the main studies in the subject of automatic verification of action-based timed simulations.

2.3.1 Timed Transition System Refinement

The action-based versions of both the strong and the weak simulations have been extended to the timed context. They are defined on a timed transition system that satisfies the executability axioms (additivity, continuity...) as already defined earlier in Chapter : Timed Systems.

Timed Strong Simulation

Given a set of labels L, two TTS Systems $A = (Q_A, Q_A^0, \rightarrow_A)$ and $C = (Q_C, Q_C^0, \rightarrow_C)$, a relation $R \subseteq Q_C \times Q_A$ is a timed strong simulation relation [25, 26] iff $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for every $(Q_C, Q_A) \in R$ and for every label $l \in L \cup \Delta$, if $q_C \xrightarrow{l} q'_C$ then there exists q'_A such that $q_A \xrightarrow{l} q'_A$ and $(q'_C, q'_A) \in R$.

This means that in the timed context, in addition to the simulation of control transitions (labeled with actions), we require as well the simulation of time transitions.

Timed Weak Simulation

Given a set of labels L_τ, two TTS Systems $A = (Q_A, Q_A^0, \rightarrow_A)$ and $C = (Q_C, Q_C^0, \rightarrow_C)$, a relation $R \subseteq Q_C \times Q_A$ is a timed weak simulation relation [24] iff $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$ and for every $(q_C, q_A) \in R$ and :

1. for every action $l \in L_\tau$, if $q_C \xrightarrow{l} q'_C$ then there exists q'_A such that $q_A \overset{l}{\Rightarrow} q'_A$ and $(q'_C, q'_A) \in R$.

2. for every $\delta \in \Delta$ if $q_C \xrightarrow{\delta} q'_C$ then there exists q'_A such that $q_A \overset{(\tau|\delta_i)^*}{\rightarrow} q'_A$ and $\Sigma\delta_i = \delta$ and $(q'_C, q'_A) \in R$.

Preserved Logic

The addition of time does not have an impact on the preserved logic. Thus, the same discussion concerning the strong (resp. weak) simulation applies in the case of the timed strong (resp. weak) simulation.

2.3.2 Timed Automata Refinement

We present two studies in the context of timed automata refinement. The simulation relation for timed automata was first defined by Alur [15] as follows :

For two timed automata $A = (\Sigma_A, Q_A, q_A^0, X_A, T_A, I_A)$ and $B = (\Sigma_B, Q_B, q_B^0, X_B, T_B, I_B)$, a timed simulation is defined as a relation R that is

defined on the corresponding semantics of the two timed automata which are two timed transition systems.

Now we discuss two studies in the context of verification of simulation between timed automata.

Timed Ready Simulation

A work that studies the refinement of timed automata appears in [65] in the context of the Uppaal [73] tool. In this work, the authors consider a slight variant of timed automata with both urgent channels and shared multi-reader/multi-writer variables. In addition to clocks, a timed automaton may then test variables in its guards and manipulate variables in its reset sets. As before, the semantics of a timed automata are given via a timed transition system, but this time it is equipped with a signature which is a tuple (R,W,IW) that describes sets of shared variables that are readable (R), writable (W), and internally writable (IW) by the transition system. The authors proceed by giving an extension of the timed weak simulation seen in the context of timed transition systems. This extension, called timed ready simulation, is proven to be monotonic w.r.t the parallel composition. In addition to the definition of the timed weak simulation, the following two clauses are added : a first condition saying that the variables of the concrete system are equal to the variables of the abstract system. A second condition saying that an urgent state in the concrete system is an urgent state in the abstract system.

The authors give an automatic technique for verifying the timed ready simulation. The verification of the timed ready simulation is restricted for fully observable (no silent actions τ) and deterministic abstract models and it may be reduced to a reachability problem (which is supported by UPPAAL). This is done via a composition with a testing automaton [9] monitoring that the behavior of the concrete system is within the bounds the abstract system and then by checking that an undesired state in the testing automaton is never reached. The testing automaton namely checks (1) that the invariant of the locations are satisfied, (2) that an abstract urgent action is also urgent in the concrete system, and (3) that the guards of the abstract system are also maintained in the concrete system. In case one of these clauses is violated, the testing automaton transits to an error state.

τ Timed Simulation

Now we discuss the work of Oudot [67]. This work studies the incremental development of component-based timed systems. The timed systems considered are timed automata (and variants). The authors give a formal definition of a τ-simulation and show how such definition is suitable for the incremental development and more specifically for the preservation of linear time properties written in MITL. Moreover, certain composability notions of the timed automata are also studied when these timed automata are built in an incremental development under the τ-simulation definition. In the following we cover briefly this work. We start by giving the definition of the τ timed simulation considered :

Let $A = (\Sigma_A, Q_A, q_A^0, X_A, T_A, I_A)$ and $C = (\Sigma_C \cup \{\tau\}, Q_C, q_C^0, X_C, T_C, I_C)$ be two TA such that $X_A \subseteq X_C$. The timed simulation is defined on the semantics of Timed Automata and is the greatest binary relation $R \subseteq Q_C \times Q_A$ such that for any $s_A = (q_A, k_A) \in Q_A$ and $s_C = (q_C, k_C) \in Q_C$ the following conditions hold :

1. $s_C \xrightarrow{a} s'_C \wedge a \in \Sigma_A \Rightarrow \exists s'_A, (s_A \xrightarrow{a} s'_A \wedge s'_C R s'_A)$ *Strict Simulation*
2. $s_C \xrightarrow{\delta} s'_C \Rightarrow \exists s'_A, (s_A \xrightarrow{\delta} s'_A \wedge s'_C \ R \ s'_A)$ *Delay Equality*
3. $s_C \xrightarrow{\tau} s'_C \Rightarrow s'_C R s_A$ *Stuttering*

Compared to the simulation definition of Alur, the timed τ simulation correspond to the weak variant that takes into consideration the silent actions.

Divergence-Sensitive and Stability-Respecting Timed τ Simulation As we have explained in the untimed context, the weak simulation does not preserve the linear time properties, mainly because of divergent paths and terminal states. This also holds in the timed context. In order to treat this problem, the definition of τ timed simulation is extended by deadlock and divergence properties. Divergence-sensitivity states that there are no non-Zeno (a non-Zeno run is a run in which time can diverge, meaning thatoudot the total time elapsed along the run goes to infinity) τ-cycles in B (a cycle in which discrete transitions are only labeled by τ is called a τ-cycle). Stability-respect means that B must not contain deadlocks which do not exist in A. This is expressed by the predicate $free$. Given a location q, $free(q)$ is the set of all valuations (of states with q as discrete part) from which a discrete transition can be taken after some delay.

Adding the following two clauses to the timed τ-simulation allows to preserve the timed linear logic MITL. The proof can be found in [67].

1, 2, 3
4. B *does not contain any non* $-$ *Zeno* $-$ τ $-$ *cycles.* *Divergence* $-$ *Sensitivity*
5. $v_B \notin free(q_B) \Rightarrow v_A \notin free(q_A)$ *Stability Respect*

Timed τ Simulation Verification The authors proceed by giving an algorithm in order to verify the τ-simulation between two timed automata. A tool called Vesta [66] is also developed that allows the automatic verification of the simulation.

2.3.3 Timed Game Automata Refinement

Another work is the one of [36, 39, 43] which led to the creation of the EC-DAR tool [1]. In this tool, a timed (weak) simulation between two TIOAs is supported. The verification is done via an on-the-fly game-based algorithm between the abstract and the concrete systems coupled with an algorithm that back-propagates the error states denoting the violation of the simulation [36].

Timed Weak Alternating Simulation Relation In the definition, it is assumed that no τ transitions exist in the abstract model. This is because permitting the τ transitions in the abstract model complicates the definition and the computation of the corresponding simulation relation. In this case any delay in the concrete model can be matched by a series of delays in the abstract model separated by τ transitions. The timed weak alternating simulation between two TGAs $A = (\Sigma, Q_A, q_A^0, X_A \backslash \tau, Inv_A, T_A)$ and $C = (Q_C, q_C^0, \Sigma, X_C, Inv_C, T_C)$ is a relation $R \subseteq Q_A \times Q_C$ such that $(q0_A, q0_C) \in R$ and for every $(q_A, q_C) \in R$ and for every observable action a the following rules are satisfied :

$$
\begin{array}{rcl}
1.\ (q_C \xrightarrow{\tau}_c q'_C) \Rightarrow ((q_A, q'_C) \in R) & & (\tau\ action) \\
2.\ (q_C \xrightarrow{a}_c q'_C) \Rightarrow \exists q'_A\ \ (q_A \xrightarrow{a}_c q'_A \wedge (q'_A, q'_C) \in R) & & (controllable) \\
3.\ (q_A \xrightarrow{a}_u q'_A) \Rightarrow \exists q'_C\ \ (q_C \xrightarrow{\tau^*}_u \xrightarrow{a}_u q'_C \wedge (q'_A, q'_C) \in R) & & (uncontrollable) \\
4.\ (q_C \xrightarrow{\delta} q'_C) \Rightarrow \exists q'_A\ \ (q_A \xrightarrow{\delta}_c q'_A \wedge (q'_A, q'_C) \in R) & & (delay)
\end{array}
$$

On-the fly verification of the simulation relation Checking the refinement is solved as a turn-based game between two players [36]. The first player, or attacker, plays outputs on S and inputs on T, whereas the second player, or defender, plays inputs on S and outputs on T. According to these rule the product of S and T is then constructed and error states are detected on-the-fly and are back-propagated. There are two kinds of error states:

1. Either the attacker may delay and violates invariants on T, which means that the defender cannot match a delay, or
2. the defender has to play a given action and cannot do so, i.e., there is a deadlock.

Restrictions on the timed game automata

1. No τ events are allowed in the abstract system. This is due to the problem of matching one delay in the implementation by series of delays and τ transitions in the specification.
2. Invariants must have the form $x \leq k$.
3. A specification automaton is a TIOA that is input-enabled meaning that in each state all the inputs should be available.
4. The implementation must satisfy the two following conditions:
   (a) Independent progress: implementations cannot get stuck in a state where it is up to the environment to induce progress. In each implementation state an output is always possible maybe after some delay.
   (b) Output urgency: if an output is available, then it cannot be delayed by the environment.

2.4 Our Simulation Choice

In this chapter, we have seen a wide range of refinement relations and sufficient conditions. Now, we motivate our refinement relation choice :

1. We need to preserve logical linear properties.
2. We need that the events of the concrete system should have been introduced in the abstraction. In addition, the events of the abstraction are not forced to occur in the concrete system. This leads to a need of simulation.
3. We need that the concrete system would be able to add local events. This means that we need the concrete to be able to have τ actions. This leads to a need of a weak simulation.
4. Being in a timed context, this leads us to a need of a timed weak simulation.
5. We need to detect deadlocks, thus an additional property needs to be verified in the simulation concerning that the concrete system may not deadlock if the abstract system does not.
6. We have seen that the weak simulation does not fully preserve linear time properties (LTL for example), and this is because of the divergent cycles (infinite cycles of τ actions) that may be present in the systems. This leads to an additional hypothesis which is the divergence sensitive property.

Here we have motivated our choice of the timed simulation we want to consider. In the next part, we will give a formal definition of this simulation.

2.5 Conclusion

We have seen in this chapter simulation definitions in both the untimed and timed context. We have seen that timed simulations are either defined at finite levels (timed automata) or at the semantic level (timed transition system). In the next chapter, we start by defining our finite timed model, give our simulation definition, give its properties and show how it can be verified.

Chapter Three

CTTS Timed Simulation

3.1 Introduction

We have seen in the previous chapters that the combinatorial explosion is generally the biggest problem of model checking-based techniques. We have also introduced the refinement as a technique that plays a major role in alleviating the risk of combinatorial explosion. Actually, instead of modeling and verifying the system as whole, the idea is to follow mostly an incremental development of the components, to verify the properties on an abstract, usually smaller system, and prove their preservation by proving the refinement between the concrete and the abstract system. This method is also coupled with the property of monotony which allows to verify a system and to replace abstract sub-components by their implementations while preserving the property. As we know, from previous chapters as well, several relations were suggested -maybe more in the untimed context than in the timed context- as refinement relations and sufficient conditions.

This chapter consists in the theoretical aspects of this PhD and is mainly focused on the definition of a timed simulation that can be used in the context of our timed systems. For this, we first give the formal definition of our considered timed systems and prove that they are compositional (Section : 3.2.5). Afterward, we define our timed weak simulation (Section : 3.3) and prove that our timed simulation guarantees trace inclusion (Section : 3.3.1) and thus preserves linear time properties (Section : 3.3.2). Furthermore, we present an automatic technique to verify it (Section : 3.4) and give its soundness and completeness proof w.r.t the mathematical simulation definition (Section : 3.5). We finish this chapter by giving a proof of the compositionality of our simulation (Section : 3.3.3).

3.2 Concrete/Abstract Systems

The semantic model of our system is the Time Transition System TTS given earlier in Chapter : Timed Systems 1.3.1. Here we recall its definition and add a composition operation.

3.2.1 Semantic Model

Definition 3.1 (Timed Transition System TTS) *Given a set of labels L_τ and let Δ be a time domain, e.g $\mathbb{R}^+$, a Timed Transition System TTS is a tuple $\langle Q, Q^0, \rightarrow\rangle$ where Q is a set of states, $Q^0 \subseteq Q$ is the set of initial states, and $\rightarrow$ is a transition relation $\subseteq Q \times (L \cup \Delta) \times Q$. We write $q \xrightarrow{l} q'$ for $(q, l, q') \in \rightarrow$.*

We define $q \xrightarrow{a^*} q' \triangleq q \xrightarrow{a} q_1 \xrightarrow{a} q_2 \cdots \xrightarrow{a} q'$ and write $q \xrightarrow{ab} q''$ if there exists q' such that $q \xrightarrow{a} q' \xrightarrow{b} q''$. We define as well $q \overset{d}{\Longrightarrow}{}^* q' \triangleq \exists q_0 \xrightarrow{\delta_0} q'_0 \xrightarrow{\tau} q_1 \xrightarrow{\delta_1} q'_1 \xrightarrow{\tau} q_2 \cdots \xrightarrow{\delta_n} q'_n$ such that $\Sigma_0^n \delta_i = d \wedge q = q_0 \wedge q' = q'_n$ and write $q \overset{d}{\Longrightarrow}{}^+ q'$ when $n > 0$ (there exists at least one τ).

TTS Composition

Definition 3.2 (TTS composition) *Given two TTSs defined over the set of labels L_τ, a set of labels $S \subseteq L$ denoting the allowed synchronizations, the TTS composition is defined as follows :*

$$\langle Q_1, Q_1^0, \rightarrow_1\rangle \underset{S}{\|} \langle Q_2, Q_2^0, \rightarrow_2\rangle = \langle Q_1 \times Q_2, Q_1^0 \times Q_2^0, \rightarrow\rangle$$

where the set of transitions $\rightarrow$ =

$$\{t_1 \odot t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2\} \cup \{t_1 \uparrow_1 \mid t_1 \in T_1\} \cup \{t_2 \uparrow_2 \mid t_2 \in T_2\}$$

is defined by the following rules :

$$\frac{t_1 : q_1 \xrightarrow{l} q'_1, t_2 : q_2 \xrightarrow{l} q'_2, l \in S \cup \Delta}{t_1 \odot t_2 : (q_1, q_2) \xrightarrow{l} (q'_1, q'_2)} \textsc{Synchronous}$$

$$\frac{t_1 : q_1 \xrightarrow{l_1} q'_1 \ , \ l_1 \notin S \cup \Delta}{t_1 \uparrow_1 : (q_1, q_2) \xrightarrow{l_1} (q'_1, q_2)} \textsc{Interleaving}_L \qquad \frac{t_2 : q_2 \xrightarrow{l_2} q'_2 \ , \ l_2 \notin S \cup \Delta}{t_2 \uparrow_2 : (q_1, q_2) \xrightarrow{l_2} (q_1, q'_2)} \textsc{Interleaving}_R$$

S is omitted when it is equal to L. This means that $\|$ is a fully synchronous composition operator while $\underset{\emptyset}{\|}$ is a fully asynchronous one and is denoted by $|||$.

TTS Properties

Definition 3.3 (No-τ-cycle) *A TTS is said to have no-τ-cycle if it does not have any infinite executions ending with transitions labeled exclusively by $\delta\tau$.*

Definition 3.4 (τ-Divergence) *Given a set of labels L_τ, a TTS $\langle Q, Q^0, \rightarrow\rangle$ is τ-divergent if for all $q \in Q$ and for all $\delta \in \Delta$, there exists q' such that $q \overset{\delta}{\Longrightarrow}{}^* q'$.*

This means that we require that time can always diverge via τ events. Namely, for all d, there always exists a $\tau\delta$ execution that advances to the date d.

Remark 3.1 *A system may be τ-Divergent and with no-τ-cycle. It suffices to reach via a finite number of τ transitions a state where time can elapse infinitely.*

Definition 3.5 (τ Zeno path) *A TTS is said to have a τ Zeno path if it has an infinite time-convergent execution sequence ($\Sigma_i^{\infty}\delta_i < \infty$) in which only τ events are executed. A TTS is τ non-Zeno if it does not have such execution sequence, that is all the execution sequences that have infinite number of τ actions are also time divergent.*

The hypothesis of τ non-zeno will be used to show inductively that a property is preserved through the elapse of time interleaved with τ transitions. The following lemma characterizes such a property.

Lemma 3.1 (τ Non-Zenoness Characterization) *We give an induction-based definition of the τ non-Zenoness of a TTS [5]. We denote as $P_\delta(s)$ a property P that is satisfied at a state s at time δ. If the TTS is τ non-zeno, we have :*

$$\frac{\overbrace{P_0(q_0)}^{(1)} \wedge \left(\begin{array}{l} \forall q \in Q\ \forall \delta_2 < \delta_1, q_0 \xrightarrow{\delta_2} q\ \wedge P_{\delta_2}(q) \Rightarrow \\ \quad \overbrace{\exists q', q \xrightarrow{\delta_1-\delta_2} q' \wedge P_{\delta_1}(q')}^{(2)}\ \vee \\ \quad \overbrace{\exists \delta_3 \in [\delta_2, \delta_1], \exists q', q \overset{\delta_3-\delta_2}{\Longrightarrow}{}^{+} q' \wedge P_{\delta_3}(q')}^{(3)} \end{array} \right)}{\exists q', q_0 \overset{\delta_1}{\Longrightarrow}{}^{*} q' \wedge P_{\delta_1}(q')}$$

The τ non-Zenoness property leads to an induction principle. Here, we say that for a property P to be true in δ, then it is sufficient to show that :

1. *P is true at the current instant (1) and,*
2. *if P is true at a given time, then P must be made true either after a time transition reaching δ (2) or after a τ transition (possibly preceded by a delay) (3).*

3.2.2 State-Related Properties

Let us recall the definition of a deadlock state. Actually, a *deadlock* state has no outgoing transitions. Formally, given set of labels L_τ, a TTS $\langle Q, Q^0, \rightarrow \rangle$ and a state $q \in Q$, we define :

$$Deadlock(q) \triangleq \forall q' \in Q, e \in L, \neg q \xrightarrow{e} q'$$

Now, in the timed context, this description is enforced by saying that a state is in a *deadlock* if even after letting time pass, no discrete transitions may be firable. This is defined [67] as :

$$TimedDeadlock(q) \triangleq \forall q' \in Q, e \in L, \delta \in \Delta, \neg q \xrightarrow{\delta} q' \xrightarrow{e} q''$$

The Timed deadlock definition is slightly weakened by saying that a state is in a weak deadlock if by doing τ events or by letting time pass, no visible events may be fired. Such definition is not surprising since upon the definition of the timed trace of a system the τ events are dropped. If follows that the traces at the deadlocked state are the same. The reason that led to the addition of the τ events stems from the definition of the timed weak simulation definition that accepts τ events.

Definition 3.6 (Weak Deadlock State) *Formally, given a TTS $\langle Q, Q^0, \rightarrow \rangle$ and a state $q \in Q$ we define :*

$$WeakDeadlock(q) \triangleq \forall q', q'' \in Q, e \in L, \delta \in \Delta, \neg q \stackrel{\delta^*}{\Longrightarrow} q' \stackrel{e}{\rightarrow} q''$$

Now we define the notion of a divergent state as a state in which time can advance infinitely.

Definition 3.7 (Divergent State) *Formally, given a TTS $\langle Q, Q^0, \rightarrow \rangle$ and a state $q \in Q$ we define :*

$$Divergent(q) \triangleq \forall \delta \exists q' \; q \stackrel{\delta}{\rightarrow} q'$$

3.2.3 Timed Executions

Definition 3.8 (Timed Execution) *A timed execution over a set of states Q and a set of labels L_τ is a finite (resp. infinite) sequence of states separated by alternating time and actions steps. Formally, it is defined as follows : $\forall i, \delta_i \in \Delta$, $\alpha_i \in L$, $q_i \in Q$*

$$(q_0 \stackrel{\delta_1 \alpha_1}{\rightarrow} q_1 \cdots q_i \stackrel{\delta_{i+1} \alpha_{i+1}}{\rightarrow} \cdots q_n \stackrel{\delta_{n+1}=\infty}{\rightarrow})_{i \in \{0..n-1\}} (resp.(q_i \stackrel{\delta_{i+1} \alpha_{i+1}}{\rightarrow})_{i \in \mathbb{N}})$$

We say that a timed execution is time divergent (resp. time convergent) if it verifies $\Sigma_i \delta_i = \infty$ (resp. $\Sigma_i \delta_i < \infty$).

Definition 3.9 (TTS Timed Execution) *A timed execution of a TTS $\langle Q, Q^0, \rightarrow \rangle$ is defined as a finite or infinite sequence of states separated by alternating time and actions steps :*

$$(q_0 \stackrel{\delta_1 \alpha_1}{\rightarrow} q_1 \cdots q_i \stackrel{\delta_{i+1} \alpha_{i+1}}{\rightarrow} \cdots q_n \stackrel{\delta_{n+1}=\infty}{\rightarrow})_{i \in \{0..n-1\}} \; (resp.(q_i \stackrel{\delta_{i+1} \alpha_{i+1}}{\rightarrow})_{i \in \mathbb{N}})$$

where, every step in the timed execution corresponds to a transition in the TTS. An initial execution is an execution starting from an initial state q_0 of the TTS and if finite, $WeakDeadlock(q_n) \wedge Divergent(q_n)$. We define $Exec(tts)$ as the set of executions of a TTS.

It is interesting to note here that a TTS execution that ends in a divergent state is time divergent. However, a time divergent execution may be without a divergent state.

Definition 3.10 (τ-Reduced TTS Execution) *The reduction of a TTS execution $(\downarrow)$ is obtained after the elimination of states and τ transitions. Here we suppose that the propositions satisfied at a state are invariant through time and silent transitions. It is defined by the following equations :*

1. $(q_0 \stackrel{d}{\rightarrow} \stackrel{(\delta\tau)^\omega}{\rightarrow}) \downarrow = q_0 \stackrel{\infty}{\rightarrow}$.
2. $(q_0 \stackrel{d}{\rightarrow} \stackrel{\tau\delta}{\rightarrow} \bar{x}) \downarrow = (q_0 \stackrel{d+\delta}{\rightarrow} \bar{x}) \downarrow$.
3. $(q_0 \stackrel{d\tau}{\rightarrow} \bar{x}) \downarrow = (q_0 \stackrel{d}{\rightarrow} \bar{x}) \downarrow$.

4. $(q_0 \stackrel{d\alpha}{\rightarrow} \bar{x})\downarrow = (q_0 \stackrel{d\alpha}{\rightarrow} \bar{x}\downarrow)$.

5. $(q_0 \stackrel{d}{\rightarrow} \stackrel{(\infty)}{\rightarrow})\downarrow = q_0 \stackrel{\infty}{\rightarrow}$.

where $\delta \in \Delta$, $e \in L$ and $\bar{x}$ is a TTS execution.

Remark 3.2 *The assumption that the states propositions are invariant through time and silent transitions makes it possible to eliminate some state observations in the trace. Concerning the case of time transitions, these transitions depend on the clocks of the TTS, but the clocks cannot be accessed at the syntactic representation of a TTS (here CTTS, see Section : 3.2.5). Thus time transitions do not change the values of the propositions. Concerning the silent transitions, this assumption is true in state-based models. Namely, in a state-based semantics, the silent action τ is assumed to loop in the same state (thus not changing the values of the propositions) [21]. However, in our state-event based context, a silent transition (τ event) may cause a change of states and thus changing the satisfied propositions. Our assumption will thus restrict a bit the properties verified on models, namely properties such as "being in state s" where s is a state having an outgoing silent transition will be forbidden.*

Definition 3.11 (Projection of a TTS Execution) *Given a valuation function $v : Q \rightarrow 2^P$ which associates to each state of the considered TTS the set of propositions satisfied by the state. Suppose that v is invariant through time and silent transitions, the projection of a TTS execution is obtained by applying the valuation function v to each state.*

3.2.4 Traces

We define a state-event based trace. This definition is later used to prove the inclusion of state-event traces and thus the preservation the State-Event-based logic properties (SE-MITL, SE-LTL).

Definition 3.12 (Timed State-Event Trace) *Given a set of propositions $\mathcal{P}$ and a set of labels L, a timed state-event trace is a finite (resp. infinite) sequence of time-steps, actions and state observations $\langle (P_0 \stackrel{\delta_1\alpha_1}{\rightarrow} P_1 \cdots P_{i+1} \stackrel{\delta_{i+1}\alpha_{i+1}}{\rightarrow} \cdots P_n \stackrel{\delta_{n+1}=\infty}{\rightarrow})_{i\in\{0..n-1\}}\rangle$ (resp. $\langle (P_i \stackrel{\delta_{i+1}\alpha_{i+1}}{\rightarrow})_{i\in\mathbb{N}}\rangle$) where $\forall i, \delta_i \in \Delta$, $\alpha_i \in L$, $P_i \subseteq \mathcal{P}$.*

We say that a timed trace is time divergent (resp. time convergent) if it verifies $\Sigma_i\delta_i = \infty$ (resp. $\Sigma_i\delta_i < \infty$). Note that in these traces, we require that the propositions are satisfied just after the occurrence of an event.

Definition 3.13 (TTS State-Event Timed Trace) *Given a valuation function $v : Q \rightarrow 2^P$ which associates to each state of the considered TTS the set of propositions satisfied by the state. Suppose that v is invariant through time and silent transitions, a (finite or infinite) state-event timed trace is accepted by a TTS $\langle Q, q_0, \rightarrow\rangle$ if it is a projection of a τ-reduced TTS execution of a divergent initial timed TTS execution. We denote by SE-Traces(T,v) the set of the state-event traces of T.*

The trace is defined from a divergent execution in order to eliminate the possibility of having τ-Zeno traces (see Definition 3.5). Otherwise, such Zeno traces would contradict our first rule in the definition of τ-Reduced

TTS Execution (Definition 3.10) that associates the symbol ∞ when $(\delta\tau)^\omega$ transitions are found.

Remark 3.3 *If the TTS has no τ-cycle (Definition : 3.3), a finite trace is a result of the τ reduction of a finite execution of the TTS.*

Definition 3.14 (TTS timed state-event traces to timed state-event sequences (TSES)) *The transformation of a TTS timed state-event trace ($\Downarrow$) is obtained by the following equations :*

1. $\bar{x} \Downarrow = (\bar{x}, 0) \Downarrow$
2. $(P_i \xrightarrow{\infty}, d) \Downarrow = (P_i, [d, \infty[)$.
3. $(P_i \xrightarrow{\delta_{i+1}\alpha_{i+1}} \bar{x}, d) \Downarrow = (P_i, [d, d+\delta[) \xrightarrow{\alpha} (\bar{x}, d+\delta) \Downarrow$.

where $\delta \in \Delta$, $\alpha \in L$ and $\bar{x}$ is a TTS timed state-event trace. For a TTS, T and a valuation function v, we will denote the set of state-event timed sequence of T by $TSES(T, v)$.

3.2.5 Constrained Time Transition System (CTTS)

We give the definition of the CTTS which is a syntactic finite representation for the TTS. We also give the properties that need to be satisfied by this finite representation in order to satisfy the assumptions made at the semantic level.

Definition 3.15 (Constrained Time Transition System) *Given a set of labels L_τ, a set of intervals $\mathbf{I}$ over the time domain Δ, a Constrained Time Transition System (CTTS) [45] is defined as $\langle Q, Q^0, T, L_\tau, \rho : T \to 2^{Q \times Q}, \lambda : T \to L_\tau, \iota : T \nrightarrow \mathbf{I}, \triangleright \subseteq T \times T \rangle$ where Q denotes the set of states, $Q^0 \subseteq Q$ is the set of initial states, T denotes the set of transitions, ρ maps each transition to a set of state couples (source,target), λ associates each transition with its label, ι associates each transition labeled with a local event with a time interval (the global events will then not be constrained) and $\triangleright$ denotes a time reset relation between two transitions. We write $t : q \xrightarrow{l} q'$ for $(q, q') \in \rho(t) \wedge l = \lambda(t)$.*

We comment on the $\triangleright$ relation. Each transition interval of a CTTS is represented implicitly by a clock at the semantic level. Whether the firing of a transition resets the clocks of the enabled transitions or not is governed by the reset relation $\triangleright$: if $t \triangleright t'$, then the firing of t resets the clock of t'.

CTTS Example In Fig 3.1, we give an example of a CTTS in which a private port is attached to an interval of time [1,2], as for the global port p it is not constrained. It is also interesting to note that the enabling of whichever of the 2 possible transitions from the initial state will reinitialize the clock of the other.

We give another example in order to illustrate the $\triangleright$ relation. In Fig. 3.2, we show two systems that are exactly the same except that they differ in the way the $\triangleright$ is defined between their transitions. Actually, in the first system of Fig. 3.2, $t_1 \triangleright t_2$ while in the second system it does not. In the first system, t_2 is never fired. This is because, at the state s_0, after exactly

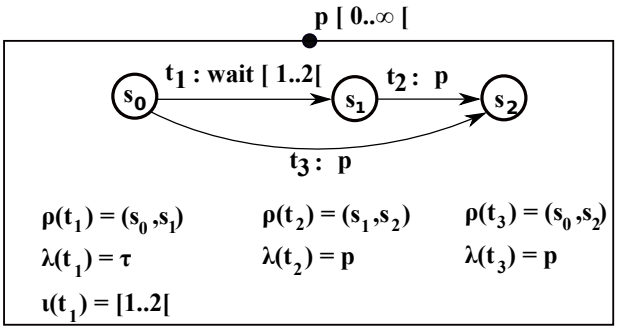


Figure 3.1 – *Example of CTTS*

1 unit of time (u.t), only the transition t_1 would be able to fire since it is associated to the time interval [1,1]. t_2 would not be able to fire since its timing constraint is not met. Now, after the firing of t_1, and since $t_1 \triangleright t_2$, the clock of t_2 is reset and the same mechanism is repeated all over again. As for the case of the second system, at 1 u.t, the transition t_1 is fired without resetting the clock of the transition t_2. So when the time reaches 2 u.t, a non-deterministic choice between t_1 and t_2 is thus possible.

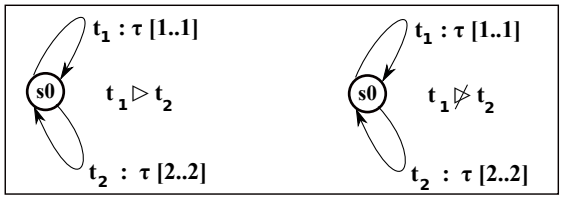


Figure 3.2 – *Reset Function of a CTTS*

CTTS Semantics as a TTS We define the semantics of a CTTS $\mathcal{T} = \langle Q, Q^0, T, L_\tau, \rho, \lambda, \iota, \triangleright \rangle$ to be a TTS $[\![\mathcal{T}]\!] = \langle Q \times (T \to \Delta), (Q^0 \times \{t : T \mapsto 0\}), \to, \alpha, \beta, \lambda \rangle$ such that $\to$ is defined by the following rules :

$$\frac{t : q \xrightarrow{l} q', \quad \begin{array}{l} v(t) \in \iota(t) \\ \wedge \forall t', v'(t') = \begin{cases} 0 & \textbf{if } (q,q') \hookrightarrow t' \vee \underline{t \triangleright t'} \\ v(t') & \textbf{else} \end{cases} \end{array}}{(q,v) \xrightarrow{l} (q', v')} \text{DIS}$$

$$\frac{}{(q,v) \xrightarrow{0} (q,v)} \text{0-DLY} \qquad \frac{\forall t \in T, q \in \mathbf{dom}(\rho(t)) \Rightarrow v(t) + \delta \in \overleftarrow{\iota(t)}}{(q,v) \xrightarrow{\delta} (q, v+\delta)} \text{DLY}$$

Figure 3.3 – *CTTS Semantics*

where $(v+\delta)(t) = v(t) + \delta$, $\overleftarrow{I} \triangleq \{x \in \Delta \mid \exists y \in \Delta, x + y \in I\}$ is the downward closure of the interval I and $(q,q') \hookrightarrow t' \triangleq q \notin \mathbf{dom}(\rho(t')) \wedge q' \in \mathbf{dom}(\rho(t'))$ means that the transition t' is newly enabled by the transition $q \to q'$ (Fig 3.4). We note here that the time properties associated to a TTS are satisfied by the CTTS semantics. Namely, it is easy to prove that the zero-delay, time determinism, time continuity and time additivity are satisfied by the semantic of the CTTS.

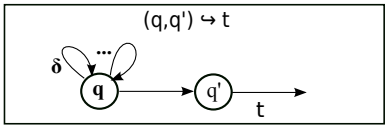


Figure 3.4 – *Newly Enabled Transition*

Definition 3.16 (CTTS Deadlock State) *A CTTS state is a deadlock state if it has no outgoing transition.*

Remark 3.4 *Based on the CTTS semantics, time diverges on deadlock states as it is not constrained by intervals associated to outgoing transitions.* [1]

Definition 3.17 (CTTS Property Satisfaction) *Given a linear temporal formula φ (written in SE-MITL, SE-LTL $\cdots$), a CTTS T and a valuation function v, we say that T satisfies φ, denoted by $(T, v) \models \varphi$, if $\forall \sigma, (\sigma \in \text{SE-Traces}([\![T]\!], v \circ \pi_1) \Rightarrow \sigma \Downarrow\models \varphi)$ where π_1 is the first projection of a couple.*

Thus, a CTTS satisfies the property φ if all its traces satisfy φ, i.e., if all its behaviors are admissible. Note here that the chosen valuation does not take into account the clocks introduced by the semantic function. Thus it is stable by the advance of time. Here we recall our assumption that state propositions are invariant through time and silent transitions and what does this assumption induce at the property verification level (see Remark : 3.2).

CTTS Composition

Here we define the composition of two CTTSs. This composition is close to the already seen operation at the TTS level. However, at the CTTS level, this composition is extended to the ι and the $\triangleright$ functions.

Definition 3.18 (CTTS Composition) *Given two CTTSs $CTTS_1 = \langle Q_1, Q_1^0, T_1, L_\tau, \rho_1, \lambda_1, \iota_1, \triangleright_1 \rangle$ and $CTTS_2 = \langle Q_2, Q_2^0, T_2, L_\tau, \rho_2, \lambda_2, \iota_2, \triangleright_2 \rangle$, a set of labels $S \subseteq L$, their composition $CTTS_1 \underset{S}{\|} CTTS_2$ is defined as $\langle Q_1 \times Q_2, Q_1^0 \times Q_2^0, T, L_\tau, \rho, \lambda, \iota, \triangleright \rangle$ where $T = \{t_1 \odot t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2\} \cup \{t_1 \uparrow_1 \mid t_1 \in T_1\} \cup \{t_2 \uparrow_2 \mid t_2 \in T_2\}$ such that :*

$$\frac{t_1 : q_1 \xrightarrow{l} q_1',\ t_2 : q_2 \xrightarrow{l} q_2',\ l \in S}{t_1 \odot t_2 : (q_1, q_2) \xrightarrow{l} (q_1', q_2')}\ \textsc{Synchronous}$$

$$\frac{t_1 : q_1 \xrightarrow{l_1} q_1',\ l_1 \notin S}{t_1 \uparrow_1 : (q_1, q_2) \xrightarrow{l_1} (q_1', q_2)}\ \textsc{Interleaving}_L \qquad \frac{t_2 : q_2 \xrightarrow{l_2} q_2',\ l_2 \notin S}{t_2 \uparrow_2 : (q_1, q_2) \xrightarrow{l_2} (q_1, q_2')}\ \textsc{Interleaving}_R$$

Figure 3.5 – *CTTS Transitions Composition*

The visible events are not time constrained. Thus, only the τ events may be associated to time intervals. ι is only defined on τ transitions. The transitions of

[1] This is different than Timed Automata where time elapses can be constrained by state invariants.

the resulting CTTS are associated to the same time intervals they had before the application of the composition operation. Formally, this is defined as :

$$\iota(t\uparrow_1) = \iota(t) \textbf{ if } \lambda(t) = \tau \qquad Time_L \qquad \iota(t\uparrow_2) = \iota(t) \textbf{ if } \lambda(t) = \tau \qquad Time_R$$

Figure 3.6 – *Interval Composition*

For $t_i, t_i' \in T_i$ and $i = \{1,2\}$, the composition of the clock-reset function $\triangleright$ is defined as :

$$\frac{t_i \triangleright t_i'}{t_i\uparrow_i \triangleright t_i'\uparrow_i}\ (1) \qquad \frac{t_i \triangleright t_i'}{t_1 \odot t_2 \triangleright t_i'\uparrow_i}\ (2) \qquad \frac{t_i \triangleright t_i'}{t_i\uparrow_i \triangleright t_1' \odot t_2'}\ (3)$$
$$\frac{t_1 \triangleright t_1'}{t_1 \odot t_2 \triangleright t_1' \odot t_2'} \qquad \frac{t_2 \triangleright t_2'}{t_1 \odot t_2 \triangleright t_1' \odot t_2'}\ (4)$$

Figure 3.7 – *Reset Function Composition*

The meaning of the reset-clock rules is that if before the composition a transition t resets the clock of another transition t', then after the composition the result transition made out of t (either by the $SYNCHRONOUS$ rule or by either one of the $INTERLEAVING$ rules) will reset the clock of the transition made out of t'. Note here that in order for the forth rule $t_1 \odot t_2 \triangleright t_1' \odot t_2'$ to be satisfied, it suffices that one of the premises $t_1 \triangleright t_1'$ or $t_2 \triangleright t_2'$ is satisfied and not necessarily both. The intuition of this rule is based on properties of the read arc of Petri Nets [6]. In the context of Petri Nets, $t \triangleright t'$ means that t has consumed a resource (absence of a read arc) needed by t'. Thus if t_1 consumes the resource used by t_1' or t_2 consumes the resource used by t_2', we will have always $t_1 \odot t_2$ consuming the resource used by $t_1' \odot t_2'$.

Compositional Semantics of a CTTS

Given two CTTSs $CTTS_1 = \langle Q_1, Q_1^0, T_1, L_\tau, \rho_1, \lambda_1, \iota_1, \triangleright_1\rangle$ and $CTTS_2 = \langle Q_2, Q_2^0, T_2, L_\tau, \rho_2, \lambda_2, \iota_2, \triangleright_2\rangle$ a set of labels $S \subseteq L$, the proof of correctness of the CTTS composition is based on proving the bisimulation between the semantics of the composition of $CTTS_1$ and $CTTS_2$ and the composition of the semantics of $CTTS_1$ and $CTTS_2$. This means that we need to prove that the composition at the semantic level (TTS) leads to a same system as the composition at the syntactic level (CTTS).

Theorem 3.1 (CTTS Compositionality) *Let $CTTS_1$ and $CTTS_2$ be two CTTS, we have :*

$$[\![CTTS_1 \underset{S}{\|} CTTS_2]\!] \simeq [\![CTTS_1]\!] \underset{S}{\|} [\![CTTS_2]\!] \qquad (Compositional\ CTTS)$$

The proof of this theorem consists in showing that each transition of the $[\![CTTS_1 \underset{S}{\|} CTTS_2]\!]$ can be found in $[\![CTTS_1]\!] \underset{S}{\|} [\![CTTS_2]\!]$ and vice versa. Based on the TTS composition (Section : 3.2.1) and the CTTS composition (Section : 3.2.5), there exists three types of transitions on the resulting system. These are a transition built by the $SYNCHRONOUS$ rule or a

transition built by either one of the symmetrical *INTERLEAVING* rules. In the following we consider these three cases. For presentation purposes, in each of the three cases we tackle the proof from left to right and then from right to left in the equivalence (*Compositional CTTS*). This proof structure can be adopted here since left to right and right to left proof trees mirror each other.

Synchronous Case

Lemma 3.2 *The transition* $((q_1, v_1), (q_2, v_2)) \xrightarrow{l}_{TTS} ((q'_1, v'_1), (q'_2, v'_2))$ *exists on* $[\![CTTS_1]\!] \underset{S}{\|} [\![CTTS_2]\!]$ *iff the transition* $((q_1, q_2), v) \xrightarrow{l}_{TTS} ((q'_1, q'_2), v')$ *exists on* $[\![CTTS_1 \underset{S}{\|} CTTS_2]\!]$*, with v and v′ defined as:*

$$\forall t, v(t) = \begin{cases} 0 & if\ t = t_1 \odot t_2 \\ v_1(t_1) & if\ t = t_1 \uparrow 1 \quad and \\ v_2(t_2) & if\ t = t_2 \uparrow 2 \end{cases}$$

$$\forall t', v'(t') = \begin{cases} 0 & \textbf{if } ((q_1, q_2), (q'_1, q'_2)) \hookrightarrow t' \vee t_1 \odot t_2 \rhd t' \\ v(t') & \textbf{else} \end{cases}$$

Proof.

$$\dfrac{t_i : q_i \xrightarrow{l}_{CTTS} q'_i \ , \quad \forall t', v'_i(t') = \begin{cases} 0 & \textbf{if } (q_i, q'_i) \hookrightarrow t' \vee t_i \rhd t' \\ v_i(t') & \textbf{else} \end{cases} \ , (i{=}1,2) \quad \forall t', q_i \in \mathbf{dom}(\rho(t')) \Rightarrow v_i(t') \in \iota(t')}{(q_1, v_1) \xrightarrow{l}_{TTS} (q'_1, v'_1) \ , \ (q_2, v_2) \xrightarrow{l}_{TTS} (q'_2, v'_2) \ , \ l \in S \cup \Delta}\ DIS$$

$$\dfrac{(q_1, v_1) \xrightarrow{l}_{TTS} (q'_1, v'_1) \ , \ (q_2, v_2) \xrightarrow{l}_{TTS} (q'_2, v'_2) \ , \ l \in S \cup \Delta}{((q_1, v_1), (q_2, v_2)) \xrightarrow{l}_{TTS} ((q'_1, v'_1), (q'_2, v'_2))}\ SYN$$

Figure 3.8 – *Building/Destructing the TTS Synchronous Product Transition*

$$\dfrac{\dfrac{t_i : q_i \xrightarrow{l}_{CTTS} q'_i \quad l \in S}{(q_1, q_2) \xrightarrow{l}_{CTTS} (q'_1, q'_2)}\ SYN, \quad \dfrac{\dfrac{}{\forall t_1, t_2, (q_1, q_2) \in \mathbf{dom}(\rho(t_1 \odot t_2)) \Rightarrow v(t_1 \odot t_2) \in \iota(t_1 \odot t_2)} \quad \dfrac{\forall t_i, q_i \in \mathbf{dom}(\rho(t_i)) \Rightarrow v_i(t_i) \in \iota(t_i)}{\forall t_i, (q_1, q_2) \in \mathbf{dom}(\rho(t_i \uparrow i)) \Rightarrow v(t_i \uparrow i) \in \iota(t_i)}\ T_L, T_R}{\forall t, (q_1, q_2) \in \mathbf{dom}(\rho(t)) \Rightarrow v(t) \in \iota(t)}, \quad \dfrac{}{\forall t', v'(t') = 0 \textbf{ if } ((q_1, q_2), (q'_1, q'_2)) \hookrightarrow t' \vee t_1 \odot t_2 \rhd t' \ v(t') \textbf{else}}\ (2),(4)}{((q_1, q_2), v) \xrightarrow{l}_{TTS} ((q'_1, q'_2), v')}\ DIS$$

Figure 3.9 – *Building/Destructing the TTS Semantics of a Synchronous Product Transition*

first implication Let $((q_1, v_1), (q_2, v_2)) \xrightarrow{l}_{TTS} ((q'_1, v'_1), (q'_2, v'_2))$ be a transition of $[\![CTTS_1]\!] \underset{S}{\|} [\![CTTS_2]\!]$.

In Fig 3.8 the rules are read from bottom to top :

- By the *SYNCHRONOUS* rule of the composition of TTS (Fig 3.5), we know that the transition $((q_1, v_1), (q_2, v_2)) \xrightarrow{l}_{TTS} ((q'_1, v'_1), (q'_2, v'_2))$ may not exist unless $(q_1, v_1) \xrightarrow{l}_{TTS} (q'_1, v'_1)$ of $[\![CTTS_1]\!]$ and $(q_2, v_2) \xrightarrow{l}_{TTS} (q'_2, v'_2)$ of $[\![CTTS_2]\!]$ both exist and that the labels may be synchronized.
- Since the transitions exist on the two TTS, then by our DIS rule (Fig 3.3) we know that these two transitions also exist on the corresponding CTTS. By applying twice (one for each transition) the DIS rule (Fig 3.3) we reach the transitions $s_1 \xrightarrow{l}_{CTTS} q'_1$ of TI_1 and $q_2 \xrightarrow{l}_{CTTS} q'_2$ of TI_2. Additionally, we reach the specification of the v_i and v'_i clocks which are defined respectively as $\forall t', v'_i(t') = \begin{cases} 0 & \textbf{if } (q_i, q'_i) \hookrightarrow t' \vee t_i \triangleright t' \\ v_i(t') & \textbf{else} \end{cases}$

Now in Fig 3.9, the rules are read from top to bottom :

- The composition of $q_1 \xrightarrow{l}_{CTTS} q'_1$ and $q_2 \xrightarrow{l}_{CTTS} q'_2$ at the CTTS level is also based on the *SYNCHRONOUS* rule (Fig 3.5) of the CTTS composition. This leads to the transition $(q_1, q_2) \xrightarrow{l}_{CTTS} (q'_1, q'_2)$.
- The clock v_i of each of the $CTTS_i$ systems is projected on the result of their composition. In the result of the composition of TTS there exists three kinds of transitions :
  1. $t_1 \odot t_2$: this leads directly to $\forall t_1, t_2, (q_1, q_2) \in \mathbf{dom}(\rho(t_1 \odot t_2)) \Rightarrow v(t_1 \odot t_2) \in \iota(t_1 \odot t_2)$ since in the synchronizing case $v(t_1 \odot t_2) = 0$.
  2. $t_i \uparrow i$ for i = (1,2) : this leads to $\forall t_i, (q_1, q_2) \in \mathbf{dom}(\rho(t_i \uparrow i)) \Rightarrow v(t_i \uparrow i) \in \iota(t_i \uparrow i)$. But based on the interval composition rules $Time_L$ and $Time_R$ (Fig 3.6) of CTTS $\iota(t_i \uparrow i) = \iota(t_i)$.

  Since the value of the clock v is known for all kinds of transitions at the result of composition of CTTS, then we can generalize this fact to every transition t as follows : $\forall t, (q_1, q_2) \in \mathbf{dom}(\rho(t)) \Rightarrow v(t) \in \iota(t)$.
- The clock v'_i is projected into the synchronization case of the composition of the CTTS.
  1. Based on rules (2) and (4) of the clock reset relation (Fig 3.7) we know that the clock transitions that were reset by v_1 and v_2 will be reset by v'.
  2. Based on the composition of the CTTS (Fig 3.5), we know that the transitions that are enabled by (q_i, q'_i) will be enabled by $(q_1, q_2), (q'_1, q'_2)$.

This leads to $\forall t', v'(t') = \begin{cases} 0 & \textbf{if } ((q_1,q_2),(q_1',q_2')) \hookrightarrow t' \vee t_1 \odot t_2 \triangleright t' \\ v(t') & \textbf{else} \end{cases}$

- Since we have the transition $(q_1,q_2) \xrightarrow{l}_{CTTS} (q_1',q_2')$ and the values of the clocks v and v′ then we can apply the definition of the DIS rule (Fig 3.3) which leads to the transition $((q_1,q_2),v) \xrightarrow{l}_{TTS} ((q_1',q_2'),v')$.

second implication Let $((q_1,q_2),v) \xrightarrow{l}_{TTS} ((q_1',q_2'),v')$ be a transition of $[\![TI_1 \underset{S}{\|} TI_2]\!]$.

In Fig 3.9, the rules are read from bottom to top :

- The application of the DIS rule (Fig 3.3) leads to $(q_1,q_2) \xrightarrow{l}_{TTS} (q_1',q_2')$ and to the valuation of the clock v as $\forall t, (q_1,q_2) \in \textbf{dom}(\rho(t)) \Rightarrow v(t) \in \iota(t)$ and the clock v′ (Synchronization) as $\forall t', v'(t') = \begin{cases} 0 & \textbf{if } ((q_1,q_2),(q_1',q_2')) \hookrightarrow t' \vee t_1 \odot t_2 \triangleright t' \\ v(t') & \textbf{else} \end{cases}$.
- Since the transition exists on the composition result of $CTTS_1$ and $CTTS_2$, then by our *SYNCHRONOUS* rule (Fig 3.5) of the composition of CTTS we know that $q_1 \xrightarrow{l}_{CTTS} q_1'$ and $q_2 \xrightarrow{l}_{CTTS} q_2'$ both exist.
- The value of the clock v are projected to their values before the composition :
  1. $\forall t_1, t_2, (q_1,q_2) \in \textbf{dom}(\rho(t_1 \odot t_2)) \Rightarrow v(t_1 \odot t_2) \in \iota(t_1 \odot t_2)$: This is the case of synchronization. In this case, $\iota(t_1 \odot t_2)$ is $[0,\infty[$ and could be projected directly without any concern about the values of the clocks.
  2. For i = (1,2), $\forall t_i, (q_1,q_2) \in \textbf{dom}(\rho(t_i \uparrow i)) \Rightarrow v(t_i \uparrow i) \in \iota(t_i \uparrow i)$: these are the cases of interleaving that are only possible on τ events. But since $\iota(t_i \uparrow i)$ is equal to $\iota(t_i)$ based on the interval composition of the CTTS (rules $Time_L$ and $Time_R$ of Fig 3.6), then the values of v_i before the composition are $\forall t_i, q_i \in \textbf{dom}(\rho(t_i)) \Rightarrow v(t_i) \in \iota(t_i)$.

In Fig 3.8, the rules are read from top to bottom :

- For $i \in \{1,2\}$, based on the composition rules (2) and (4) of the clock reset relation (Fig 3.7), the values of the clock v' leads to the clocks v_i where $\forall t', v_i'(t') = \begin{cases} 0 & \textbf{if } (q_i,q_i') \hookrightarrow t' \vee t_i \triangleright t' \\ v_i(t') & \textbf{else} \end{cases}$
- Having the two transitions at the CTTS level along with the valuation of v_i and v_i', the DIS rule (Fig 3.3) is applied which leads into $(q_1,v_1) \xrightarrow{l}_{TTS} (q_1',v_1')$ of $[\![TI_1]\!]$ and $(q_2,v_2) \xrightarrow{l}_{TTS} (q_2',v_2')$ of $[\![TI_2]\!]$.
- Now the composition of these two transitions at the TTS level is also based on the *SYNCHRONOUS* rule of the CTTS composition (Fig 3.5). This leads into the transition $((q_1,v_1),(q_2,v_2) \xrightarrow{l}_{TTS} ((q_1',v_1'),(q_2',v_2'))$ which happens to be our awaited conclusion.

Since the two implications hold we conclude that $((q_1, v_1), (q_2, v_2) \xrightarrow{l}_{TTS} ((q_1', v_1'), (q_2', v_2')) \simeq ((q_1, q_2), v) \xrightarrow{l}_{TTS} ((q_1', q_2'), v')$. □

Left Interleaving Case

Lemma 3.3 *The transition* $((q_1, v_1), (q_2, v_2)) \xrightarrow{l_1}_{TTS} ((q_1', v_1'), (q_2, v_2))$ *exists on* $[\![TI_1]\!] \underset{S}{\|} [\![TI_2]\!]$ *iff the transition* $((q_1, q_2), v) \xrightarrow{l_1}_{TTS} ((q_1', q_2), v')$ *exists on* $[\![TI_1 \underset{S}{\|} TI_2]\!]$, *with v and v′ defined as:*

$$\forall t, v(t) = \begin{cases} 0 & if\ t = t_1 \odot t_2 \\ v_1(t_1) & if\ t = t_1 \uparrow 1 \\ v_2(t_2) & if\ t = t_2 \uparrow 2 \end{cases}$$

$$and\ \forall t', v'(t') = \begin{cases} 0 & \textbf{if}\ ((q_1, q_2), (q_1', q_2)) \hookrightarrow t' \vee t_1 \uparrow 1 \triangleright t' \\ v(t') & \textbf{else} \end{cases}$$

Proof.

$$\dfrac{\dfrac{t_1 : q_1 \xrightarrow{l_1}_{CTTS} q_1' \ ,\ \begin{array}{c}\forall t', q_1 \in \mathbf{dom}(\rho(t')) \Rightarrow v_1(t') \in \iota(t') \\ \forall t', v_1'(t') = \begin{cases} 0 & \textbf{if}\ (q_1, q_1') \hookrightarrow t' \vee t_1 \triangleright t' \\ v_1(t') & \textbf{else} \end{cases}\end{array}}{(q_1, v_1) \xrightarrow{l_1}_{TTS} (q_1', v_1')\ ,\ l_1 \notin S \cup \Delta}\ DIS\ ,\ (q_2, v_2) = (q_2', v_2')}{((q_1, v_1), (q_2, v_2)) \xrightarrow{l_1}_{TTS} ((q_1', v_1'), (q_2, v_2))}\ INT_L$$

Figure 3.10 – *Building/Destructing the TTS Left Interleaving Product Transition*

$$\dfrac{\dfrac{t_1 : q_1 \xrightarrow{l_1}_{CTTS} q_1' \quad l_1 \notin S}{(q_1, q_2) \xrightarrow{l_1}_{CTTS} (q_1', q_2)}\ INT_L\ ,\ \dfrac{\dfrac{\dfrac{\forall t_1, t_2, (q_1, q_2) \in \mathbf{dom}(\rho(t_1 \odot t_2)) \Rightarrow v(t_1 \odot t_2) \in \iota(t_1 \odot t_2)}{\forall t_i, q_i \in \mathbf{dom}(\rho(t_i)) \Rightarrow v_i(t_i) \in \iota(t_i)}\ T_L, T_R}{\forall t_i, (q_1, q_2) \in \mathbf{dom}(\rho(t_i \uparrow i)) \Rightarrow v(t_i \uparrow i) \in \iota(t_i)}}{\forall t, (q_1, q_2) \in \mathbf{dom}(\rho(t)) \Rightarrow v(t) \in \iota(t)}\ ,\ \dfrac{\forall t', v'(t') = 0\ \textbf{if}\ ((q_1, q_2), (q_1', q_2)) \hookrightarrow t' \vee t_1 \uparrow 1 \triangleright t'\ v(t')\textbf{else}}{}\ (1),(3)}{((q_1, q_2), v) \xrightarrow{l_1}_{TTS} ((q_1', q_2), v')}\ DIS$$

Figure 3.11 – *Building/Destructing the TTS Semantics of a Left Interleaving Product Transition*

The proof of Lemma 3.3 is similar to Lemma 3.2. It suffices to reason with the $Interleaving_L$, rules (1) and (3) of the clock-reset. □

Right Interleaving Case

Lemma 3.4 *The transition* $((q_1, v_1), (q_2, v_2)) \xrightarrow{l_2}_{TTS} ((q_1, v_1), (q_2', v_2'))$*exists*

on $[\![TI_1]\!] \underset{S}{\|} [\![TI_2]\!]$ *iff the transition* $((q_1, q_2), v) \xrightarrow{l_2}_{TTS} ((q_1, q_2'), v')$ *exists on* $[\![TI_1 \underset{S}{\|} TI_2]\!]$*, with v and v' are defined as:*

$$\forall t, v(t) = \begin{cases} 0 & if\ t = t_1 \odot t_2 \\ v_1(t_1) & if\ t = t_1 \uparrow 1 \quad and \\ v_2(t_2) & if\ t = t_2 \uparrow 2 \end{cases}$$

$$\forall t', v'(t') = \begin{cases} 0 & \textbf{if } ((q_1, q_2), (q_1, q_2')) \hookrightarrow t' \vee t_2 \uparrow 2 \triangleright t' \\ v(t') & \textbf{else} \end{cases}$$

Proof. This is the symmetrical case of the left interleaving case. For the proof it suffices to reason with the (1) and (3) of the clock-reset rules. □

Compositionality of the CTTS Semantics. Having Lemma 3.4, Lemma 3.3 and Lemma 3.2, then $[\![TI_1 \underset{S}{\|} TI_2]\!] \simeq [\![TI_1]\!] \underset{S}{\|} [\![TI_2]\!]$ □

CTTS Properties

Property 3.1 (Bisimilar States) *Given a CTTS T, two states* (q, v) *and* (q, v') *in* $[\![T]\!]$ *that associate the same valuations (w.r.t v and v') to enabled* τ *transitions are bisimilar.*

This means that the states (q, v) and (q, v') can only differ in the valuation associated to τ transitions that are not enabled in q. However the valuations of clocks associated to transitions labeled by visible events can differ because they are unconstrained.

Proof of Bisimilarity. To prove this property we introduce the relation R defined as :

$$R((q_c, v_c), (q_a, v_a)) \equiv$$
$$(q_c = q_a \wedge (\forall t, \lambda(t) = \tau \wedge q_c \in \mathbf{dom}(\rho(t)) \Rightarrow v_c(t) = v_a(t)))$$

The proof of this property consists in showing its preservation by the three kinds of transitions (visible, τ and δ) available on a TTS (TTS Definition 3.1). Suppose $R(q_1, v_1), (q_1, v_1')$. Suppose that $(q_1, v_1) \xrightarrow{l} (q_2, v_2)$ where $\alpha \in \{\alpha, \tau, \delta\}$. We show that there exists a state (q_2, v_2') such that $(q_1, v_1') \xrightarrow{l} (q_2, v_2')$ and $R((q_2, v_2), (q_2, v_2'))$.

1. Visible Event α : a visible event of CTTS is not time constrained, it can thus happen unconditionally in q_1 (at both the concrete and the abstract levels). The firing of the transition leads in both cases to the state q_2. The effect of the transition firing on the clocks valuations is to reset a subset of clocks. Since resetting just induces additional clock equalities w.r.t v_2 and v_2', the relation R is preserved : $R((q_2, v_2), (q_2, v_2'))$.

2. τ Event : a τ transition happens when the transition's clock satisfies its corresponding time interval constraint. By the definition of R, the clocks for the τ transition have the same valuations (w.r.t v_1 and v_1'). The firing of τ transition in both of (q_1, v_1) and (q_1, v_1') happens at the same time and leads in both cases to the state q_2 where a subset of clocks is reset. It follows that the relation R is preserved : $R((q_2, v_2), (q_2, v_2'))$.

3. Delay : a δ transition is fired if after adding δ to the clocks of the enabled τ transitions, their maximal bounds are not reached. This condition is satisfied at the same time at both the abstract and the concrete level in q_1. Adding δ to already equal clock values preserves the equality, thus R is preserved.

Thanks to Definition 3.1, we conclude that the systems are bisimilar. □

Definition 3.19 (1-τ) *A CTTS is called 1-τ if it does not have two successive τ actions. Formally, $t : q \xrightarrow{\tau} q' \wedge t' : q' \xrightarrow{l} q'' \Rightarrow l \neq \tau$.*

We define now the following restriction on a CTTS. This restriction is used when verifying the time weak simulation between two CTTSs.

Definition 3.20 (Upper Closure) *A CTTS is called upper bounded closed if its right bounded intervals are right closed .*

Remark 3.5 *The CTTS of Fig 3.1 is not upper bounded closed since the interval associated to the transition t_1 is right opened.*

Property 3.2 (Upper Closure Preservation) *Given two upper bounded closed $CTTS_1$ and $CTTS_2$ and a set of synchronization labels S, their composition $CTTS_1 \underset{S}{\|} CTTS_2$ is also upper bounded closed.*

3.3 Timed Weak Simulation

We define our timed weak simulation relation between the TTSs.

Definition 3.21 (State-Event Timed Weak Simulation) *Given the set of labels L_τ and two TTS $\mathcal{A} = \langle Q_a, Q_a^0, \rightarrow_a \rangle$ and $\mathcal{C} = \langle Q_c, Q_c^0, \rightarrow_c \rangle$ defined over L_τ, a set of propositions P and two valuation functions $v_c : Q_c \rightarrow 2^P$ and $v_a : Q_a \rightarrow 2^P$, a simulation between them $\precsim$ is the largest relation such that :*

$$\forall q_c\ q_a, q_c \precsim q_a \Rightarrow$$

V. $v_c(q_c) = v_a(q_a)$ *(Valuations)*

E. $\forall q_c'\ e, q_c \xrightarrow{e}_c q_c' \Rightarrow \exists q_a', q_a \xrightarrow{\tau^* e}_a q_a' \wedge q_c' \precsim q_a'$ *(Visible Events)*

T. $\forall q_c', q_c \xrightarrow{\tau}_c q_c' \Rightarrow q_c' \precsim q_a$ *(τ Events)*

D. $\forall q_c'\ \delta, q_c \xrightarrow{\delta}_c q_c' \Rightarrow \exists q_a', q_a \xRightarrow{\delta^*}_a q_a' \wedge q_c' \precsim q_a'$ *(Delay)*

We say that $(\mathcal{C}, v_c) \precsim (\mathcal{A}, v_a)$ if $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$.

Remark that unlike other timed simulations definitions already seen in Chapter : Timed Systems, in our timed weak simulation definition, the abstract system may have τ events. This is needed in our FIACRE context, since in FIACRE systems timing constraints are added to local events. Allowing abstract τ events will thus enables us to add time to the abstract system.

In order to preserve the linear time properties, additional clauses are added to timed weak simulation. This leads to the State-Event Deadlock-Sensitive (DS) Timed Weak Simulation. We will be denoting this relation by $\precsim_{DS}$.

Definition 3.22 (Deadlock-Sensitive Timed Weak Simulation) *Given the set of labels L_τ and two TTS Systems $\mathcal{A} = \langle Q_a, Q_a^0, \rightarrow_a \rangle$ and $\mathcal{C} = \langle Q_c, Q_c^0, \rightarrow_c \rangle$ defined over the same alphabet, a set of propositions and two valuation functions $v_c : Q_c \rightarrow 2^P$ and $v_a : Q_a \rightarrow 2^P$, a simulation between them $\precsim_{DS}$ is the largest relation such that :*

$$\forall q_c\ q_a,\ q_c \precsim_{DS} q_a \Rightarrow$$

V $\wedge$ **E** $\wedge$ **T** $\wedge$ **D**
N_C. *$\mathcal{C}$ has no-τ-cycle* ($Definition$ 3.3).
D. $\forall q'_a\ e, q_a \xrightarrow{e}_a q'_a \Rightarrow \exists q'_c, q_c \xrightarrow{e}_c q'_c \wedge v_c(q'_c) = v_a(q'_a)$ $(Deadlock)$

We say that $(C, v_c) \precsim_{DS} (A, v_a)$ if $\forall q_C^0 \in Q_C^0, \exists q_A^0 \in Q_A^0$ such that $(q_C^0, q_A^0) \in R$.
The fifth clause (**N_C**) expresses that the refinement does not authorize infinite sequences of τ transitions in the concrete system. This means that there always must be a way out of these cycles, by doing a visible event. The last clause expresses that the concrete system C must not contain deadlocks which do not exist in the abstract system A. Remark that in order to preserve deadlocks and to have a compositional definition (See Theorem 3.3.3), we require that the existance of an abstract visible transition implies its existance at the concrete level with the same label and compatible valuations.

3.3.1 Traces Inclusion

It has been known, for a couple of decades now that simulation relations are a sufficient condition for traces inclusions. Actually, the reason behind the definition of timed simulation relations, is the non-decidability of the language inclusion for timed models like timed automata [15]. Depending on the chosen definitions of the timed models, its traces and the timed simulation relation, the result concerning the traces inclusion may differ. Here we show that our chosen timed weak simulation is a sufficient condition for timed traces inclusion.

Theorem 3.2 (SE-Traces Inclusion) *Given a set of propositions $\mathcal{P}$, two CTTSs, A and C, and two valuation functions $v_c : Q_c \rightarrow 2^{\mathcal{P}}$ and $v_a : Q_a \rightarrow 2^{\mathcal{P}}$, if $(C, v_c) \precsim_{DS} (A, v_a)$ then SE-Traces$([\![C]\!], v_c o \pi_1) \subseteq$ SE-Traces$([\![A]\!], v_a o \pi_1)$.*

Proof. This proof is a standard one in the non-timed context. Let A, C be two CTTS, such that $(C, v_c) \precsim_{DS} (A, v_a)$. We prove that SE-Traces$([\![C]\!], v_c o \pi_1) \subseteq$ SE-Traces$([\![A]\!], v_a o \pi_1)$. Let t_c be a trace $\in$ SE-Traces$([\![C]\!], v_c o \pi_1)$. We show by a step-by-step construction that t_c is also a trace $\in$ SE-Traces$([\![A]\!], v_a o \pi_1)$.

1. Based on our hypothesis that $(C, v_c) \precsim_{DS} (A, v_a)$, we show that we can build an execution in the abstract system, denoted as $exec_a$, that has the trace t_c as a projection. Suppose that the initial state q_c^0 of $exec(t_c)$ and the initial state q_a^0 of $exec_a$ are in a simulation relation $q_c^0 \precsim_{DS} q_a^0$. Inductively we have :

   (a) $v_c(q_c^0) = v_a(q_a^0)$.

(b) For all $q_c \in Q_c$,$q_a \in Q_a$, assume that α is a discrete action from $q_c \xrightarrow{\alpha}_c q'_c$: in this case, based on the *Visible Events* rule of $\precsim_{DS}$ we know that $\exists q'_a\ \alpha, q_a \xrightarrow{\tau^*\alpha}_a q'_a$ and $q'_c \precsim_{DS} q'_a$. This means that α is reachable from q_a and thus appears in $exec_a$ prefixed by τ^*.

(c) For all $q_c \in Q_c$,$q_a \in Q_a$, assume that δ is a delay from $q_c \xrightarrow{\delta}_c q'_c$: in this case, based on the *Delay* rule of $\precsim_{DS}$ we know that $\exists q'_a, q_a \xrightarrow{(\tau|\delta_i)^*}_a q'_a \wedge \Sigma\delta_i = \delta \wedge q'_c \precsim_{DS} q'_a$. This also means that δ can elapse from q_a and thus appear in $exec_a$ in the form of $(\tau|\delta_i)^*$.

(d) For all $q_c \in Q_c$,$q_a \in Q_a$, assume that τ is a local event $q_c \xrightarrow{\tau}_c q'_c$: in this case, based on the τ *Events* rule of $\precsim_{DS}$ we know that $q'_c \precsim_{DS} q_a$.

If the concrete trace t_c is infinite, the corresponding execution is also infinite. In this case, the preceding rules build an infinite abstract execution with the same trace. Otherwise, t_c is finite. Based on our hypothesis that the C has no τ-cycle, then we know that the execution corresponding to the trace t_c, denoted by $exec(t_c)$ is also finite. This means that it ends in a concrete deadlock state. Given the property **D** of our simulation definition (Definition 3.22), the corresponding abstract state is also in deadlock.

We have shown that there exists an abstract execution $exec_a$ that is in a simulation relation with $exec(t_c)$ such that it only differs in its τ events. But we know that in order to build a corresponding trace of an execution, we first need to apply the τ-reduction transformation rule (Definition 3.10), followed by applying the projection rule. Hence, the τ events are dropped when transforming the execution into a corresponding τ-reduced execution. Applying the projection rule to a τ-reduced version of $exec_a$ leads into a trace t_a which is the same as the projection of the τ-reduced version of $exec_c$. Thus t_c coincide with t_a.

□

3.3.2 Property Preservation

As we have shown, our simulation relation is a sufficient condition for traces inclusion. However, this is just one piece of the puzzle. The complementary result is the relation between the traces inclusion and the preservation of linear time properties. A strong relation between these last two exists. Actually, this is a standard result in the untimed context. Here we extend this result and its proof in our context. The original proof can be found in [21].

Theorem 3.3 (Property Preservation) *Given* $A = \langle Q_a, Q_a^0, T_a, L_\tau, \rho_a, \lambda_a, \iota_a, \triangleright_a \rangle$ *and* $C = \langle Q_c, Q_c^0, T_c, L_\tau, \rho_c, \lambda_c, \iota_c, \triangleright_c \rangle$ *with two valuation functions* $v_c : Q_c \rightarrow 2^{\mathcal{P}}$ *and* $v_a : Q_a \rightarrow 2^{\mathcal{P}}$ *saying that SE-Traces*$(\llbracket C \rrbracket, v_c o \pi_1) \subseteq$ *SE-Traces*$(\llbracket A \rrbracket, v_a o \pi_1)$ *is equivalent to saying that for any linear time property*

φ (expressed as a set of TSET): $(A, v_a) \models \varphi \Rightarrow (C, v_c) \models \varphi$.

$$\underbrace{\textit{SE-Traces}([\![C]\!], v_c o \pi_1) \subseteq \textit{SE-Traces}([\![A]\!], v_a o \pi_1)}_{traceInc} \Leftrightarrow \underbrace{(A, v_a) \models \varphi \Rightarrow (C, v_c) \models \varphi}_{LTPres}$$

Proof. For two CTTS, $A = \langle Q_a, Q_a^0, T_a, L, \rho_a, \lambda_a, \iota_a, \triangleright_a \rangle$ and $C = \langle Q_c, Q_c^0, T_c, L, \rho_c, \lambda_c, \iota_c, \triangleright_c \rangle$ with two valuation functions $v_c : Q_c \rightarrow 2^{\mathcal{P}}$ and $v_a : Q_a \rightarrow 2^{\mathcal{P}}$:

$(traceInc) \Rightarrow (LTPres)$:
Assume that SE-Traces$([\![C]\!], v_c o \pi_1) \subseteq$ SE-Traces$([\![A]\!], v_a o \pi_1)$ and let φ be an LT property such that $A \models \varphi$. From Definition 3.17 it follows that $\forall \sigma, (\sigma \in$ SE-Traces$([\![A]\!], v_a o \pi_1) \Rightarrow \sigma \Downarrow \models \varphi)$. Since by hypothesis, SE-Traces$([\![C]\!], v_c o \pi_1) \subseteq$ SE-Traces$([\![A]\!], v_a o \pi_1)$, it now follows that $\forall \sigma, (\sigma \in$ SE-Traces$([\![C]\!], v_c o \pi_1) \Rightarrow \sigma \Downarrow \models \varphi$. By Definition 3.17 it follows that $(C, v_c) \models \varphi$.

$(LTPres) \Rightarrow (traceInc)$:
Assume that for all LT properties it holds that: $(A, v_a) \models \varphi$ implies $(C, v_c) \models \varphi$. Let $\varphi = TSET([\![A]\!], v_a o \pi_1)$. Obviously, $(A, v_a) \models \varphi$. By assumption, $(C, v_c) \models \varphi$. Hence, $TSET([\![C]\!], v_c o \pi_1) \subseteq \varphi (= TSET([\![A]\!], v_a o \pi_1))$. It follows that SE-Traces$([\![C]\!], v_c o \pi_1) \subseteq$ SE-Traces$([\![A]\!], v_a o \pi_1)$. □

3.3.3 Compositional Simulation

Our timed weak simulation relation preserves the compositionality of the CTTS. By compositionality we mean that the parallel operator is monotonic w.r.t our timed simulation. Given a component C inside of a composition $C \parallel C_1 \parallel C_2 ... C_n$ the monotony of $\parallel$ is informally described as : if C is replaced by a component C' such that C' simulates C, then $C' \parallel C_1 \parallel C_2 ... C_n$ simulates $C \parallel C_1 \parallel C_2 ... C_n$. Such a result is then one of the cornerstones of incremental development and of modular verification. We give in the following the theorem of the simulation compositionality.

Definition 3.23 (Valued CTTS Composition) *Given two CTTS, $CTTS_1 = \langle Q_1, Q_1^0, T_1, L_\tau, \rho_1, \lambda_1, \iota_1, \triangleright_1 \rangle$, $CTTS_2 = \langle Q_2, Q_2^0, T_2, L_\tau, \rho_2, \lambda_2, \iota_2, \triangleright_2 \rangle$, with the valuation functions $v_1 : Q_1 \rightarrow 2^{\mathcal{P}_1}$, $v_2 : Q_2 \rightarrow 2^{\mathcal{P}_2}$ and the set of labels S, we define their valued composition as the couple $(CTTS, v)$ where :*

- *$CTTS$ is the restriction of $CTTS_1 \underset{S}{\|} CTTS_2$ to coherent states belonging to the following set :*

$$I = \{(q_1, q_2) \in Q_1 \times Q_2 \mid v_1(q_1) \cap \mathcal{P}_2 = v_2(q_2) \cap \mathcal{P}_1\}$$

- $v(q_1, q_2) = v_1(q_1) \cup v_2(q_2)$.

We denote their composition by $(CTTS_1, v_1) \underset{S}{\|} (CTTS_2, v_2)$.

Coherent states are product states of which two projections do not have contradictory properties : if p is a mutual property of $\mathcal{P}_1$ and $\mathcal{P}_2$, the two projections of the states must agree on the truth value of p defined by v_1 and v_2.

Theorem 3.4 (Simulation Compositionality) *Given the valued CTTSs* (A_1, v_{a_1}),(A_2, v_{a_2}),(C_1, v_{c_1}) *and* (A_2, v_{c_2}) *and the set of labels S. Let* (A, v_a) *and* (C, v_c) *be the composition of respectively the abstract and concrete CTTSs, we have :*

$$\frac{(C_1, v_{c_1}) \precsim_{DS} (A_1, v_{a_1}) \quad (C_2, v_{c_2}) \precsim_{DS} (A_2, v_{a_2})}{(C, v_c) \precsim_{DS} (A, v_a)}$$

Proof. Given the CTTS, $A_1 = \langle Q_{a_1}, Q^0_{a_1}, T_{a_1}, L_\tau, \rho_{a_1}, \lambda_{a_1}, \iota_{a_1}, \triangleright_{a_1} \rangle$, $A_2 = \langle Q_{a_2}, Q^0_{a_2}, T_{a_2}, L_\tau, \rho_{a_2}, \lambda_{a_2}, \iota_{a_2}, \triangleright_{a_2} \rangle$, $C_1 = \langle Q_{c_1}, Q^0_{c_1}, T_{c_1}, L_\tau, \rho_{c_1}, \lambda_{c_1}, \iota_{c_1}, \triangleright_{c_1} \rangle$ and $C_2 = \langle Q_{c_2}, Q^0_{c_2}, T_{c_2}, L_\tau, \rho_{c_2}, \lambda_{c_2}, \iota_{c_2}, \triangleright_{c_2} \rangle$ with the valuation functions $v_{c_1} : Q_{c_1} \to 2^{\mathcal{P}_1}$, $v_{c_2} : Q_{c_2} \to 2^{\mathcal{P}_2}$, $v_{a_1} : Q_{a_1} \to 2^{\mathcal{P}_1}$, and $v_{a_2} : Q_{a_2} \to 2^{\mathcal{P}_2}$ and the set of labels S, we prove that :

$$(C_1, v_{c_1}) \precsim_{DS} (A_1, v_{a_1}) \wedge (C_2, v_{c_2}) \precsim_{DS} (A_2, v_{a_2}) \Rightarrow$$
$$(C, v_c) \precsim_{DS} (A, v_a)$$

where :

- C is the restriction of $(C_1, v_{c_1}) \underset{S}{\|} (C_2, v_{c_2})$ to coherent states of $I_c = \{(q_{c_1}, q_{c_2}) \in Q_{c_1} \times Q_{c_2} \mid v_{c_1}(q_{c_1}) \cap \mathcal{P}_2 = v_{c_2}(q_{c_2}) \cap \mathcal{P}_1\}$.
- $v_c(q_{c_1}, q_{c_2}) = v_{c_1}(q_{c_1}) \cup v_{c_2}(q_{c_2})$
- A is the restriction of $(A_1, v_{a_1}) \underset{S}{\|} (A_2, v_{a_2})$ to coherent states of $I_a = \{(q_{a_1}, q_{a_2}) \in Q_{a_1} \times Q_{a_2} \mid v_{a_1}(q_{a_1}) \cap \mathcal{P}_2 = v_{a_2}(q_{a_2}) \cap \mathcal{P}_1\}$.
- $v_a(q_{a_1}, q_{a_2}) = v_{a_1}(q_{a_1}) \cup v_{a_2}(q_{a_2})$

We define the relation R between the states of (C, v_c) and (A, v_a) as $R((q_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2})) = q_{c_1} \precsim_{DS} q_{a_1} \wedge q_{c_2} \precsim_{DS} q_{a_2}$. We verify whether R satisfies the clauses of $\precsim_{DS}$.

Given $(q_{c_1}, q_{c_2}) \in I_c$ and $(q_{a_1}, q_{a_2}) \in I_a$ such that $R((q_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2}))$:

1. **Propositions** : By the hypothesis $(C_1, v_{c_1}) \precsim_{DS} (A_1, v_{a_1})$ and $(C_2, v_{c_2}) \precsim_{DS} (A_2, v_{a_2})$ we know respectively that $v_{c_1}(q_{c_1}) = v_{a_1}(q_{a_1})$ and $v_{c_2}(q_{c_2}) = v_{a_2}(q_{a_2})$. It follows that $v_{c_1}(q_{c_1}) \cup v_{c_2}(q_{c_2}) = v_{a_1}(q_{a_1}) \cup v_{a_2}(q_{a_2})$. Thus by definition we reach $v_c(q_{c_1}, q_{c_2}) = v_a(q_{a_1}, q_{a_2})$. Thus R satisfies the first clause of $\precsim_{DS}$.

   Let t_c be a transition in $C_1 \underset{S}{\|} C_2$ where $\lambda \in \{\alpha, \tau, \delta\}$.

2. **Visible Events :** Let $\lambda = \alpha$ be a discrete action. Three cases need to be investigated which belong to the synchronous and the two interleaving cases of the CTTS composition (Fig 3.5) :

   (a) Synchronous case : since $t_c : (q_{c_1}, q_{c_2}) \xrightarrow{\alpha} (q'_{c_1}, q'_{c_2})$ is present in the composition (C, v_c) then based on the *SYNCHRONOUS* rule of the composition, $t_{c_1} : q_{c_1} \xrightarrow{\alpha} q'_{c_1}$ and $t_{c_2} : q_{c_2} \xrightarrow{\alpha} q'_{c_2}$ also exist at each side of the composition C_1 and C_2. Having by hypothesis $q_{c_1} \precsim_{DS} q_{a_1}$ and $q_{c_2} \precsim_{DS} q_{a_2}$ then by applying the rule *Visible Events* of $\precsim_{DS}$ on each of t_{c_1} and t_{c_2} we conclude

respectively that there exists q'_{a_1} such that $t_{a_1} : q_{a_1} \xrightarrow{\tau_a^* \alpha} q'_{a_1} \wedge q'_{c_1} \precsim_{DS} q'_{a_1}$ and that there exists q'_{a_2} such that $t_{a_2} : q_{a_2} \xrightarrow{\tau_a^* \alpha} q'_{a_2} \wedge q'_{c_2} \precsim_{DS} q'_{a_2}$. Now, the composition of t_{a_1} and t_{a_2} would lead to $t_a : (q_{a_1}, q_{a_2}) \xrightarrow{\tau_a^* \alpha} (q'_{a_1}, q'_{a_2})$. Having $q'_{c_1} \precsim_{DS} q'_{a_1}$ and $q'_{c_2} \precsim_{DS} q'_{a_2}$ then by our definition we reach $((q'_{c_1}, q'_{c_2}), (q'_{a_1}, q'_{a_2})) \in R$ and R verifies the second clause of $\precsim_{DS}$.

We are left with proving that (q'_{a_1}, q'_{a_2}) is coherent, meaning that it respects the invariant I_a. Particularly, we need to show that $v_{a_1}(q'_{a_1}) \cap \mathcal{P}_2 = v_{a_2}(q'_{a_2}) \cap \mathcal{P}_1$. We know, that (q'_{c_1}, q'_{c_2}) is coherent, thus $v_{c_1}(q'_{c_1}) \cap \mathcal{P}_2 = v_{c_2}(q'_{c_2}) \cap \mathcal{P}_1$. Since $q'_{c_1} \precsim_{DS} q'_{a_1}$ and $q'_{c_2} \precsim_{DS} q'_{a_2}$, then by the rule **V** of $\precsim_{DS}$, we reach $v_{c_1}(q'_{c_1}) = v_{a_1}(q'_{a_1})$ and $v_{c_2}(q'_{c_2}) = v_{a_2}(q'_{a_2})$. It follows that (q'_{a_1}, q'_{a_2}) is coherent.

(b) Left Interleaving : in the left interleaving case, the transition $t_c : (q_{c_1}, q_{c_2}) \xrightarrow{\alpha} (q'_{c_1}, q_{c_2})$ is present in the composition (C, v_c) as a result of the firing of the left-side transition of the composition and $\alpha \notin S$. Based on the $INTERLEAVING_L$ rule of the composition we can find $t_{c_1} : q_{c_1} \xrightarrow{\alpha} q'_{c_1}$ while C_2 remains at q_{c_2} state. Based on our hypothesis, then by using the rule *Visible Events* of $\precsim_{DS}$ we can find q'_{a_1} such that $t_{a_1} : q_{a_1} \xrightarrow{\tau_a^* \alpha} q'_{a_1} \wedge q'_{c_1} \precsim_{DS} q'_{a_1}$. The transition t_a in the composition is then built as $t_a : (q_{a_1}, q_{a_2}) \xrightarrow{\tau_a^* \alpha} (q'_{a_1}, q_{a_2})$. Having $q'_{c_1} \precsim_{DS} q'_{a_1}$ and $q_{c_2} \precsim_{DS} q_{a_2}$ then by our definition we reach $((q'_{c_1}, q_{c_2}), (q'_{a_1}, q_{a_2})) \in R$ and R verifies the second clause of $\precsim_{DS}$.

Finally, by following a similar reasoning as in the synchronous case, we show that (q'_{a_1}, q_{a_2}) is coherent.

(c) Right Interleaving : this is the symmetrical case of the left interleaving case.

3. τ **Events :** λ is a local action τ. Two cases need to be investigated which belong the two interleaving cases.

   (a) Left τ : in the left τ case, the transition $t_c : (q_{c_1}, q_{c_2}) \xrightarrow{\tau} (q'_{c_1}, q_{c_2})$ is present in the composition (C, v_c) as a result of the firing of the left-side transition of the composition. Based on the $INTERLEAVING_L$ rule of the composition we can find $t_{c_1} : q_{c_1} \xrightarrow{\tau} q'_{c_1}$ while C_2 remains at q_{c_2} state. Based on our hypothesis, then by using the rule τ *Events* of $\precsim_{DS}$ we have $q'_{c_1} \precsim_{DS} q_{a_1}$. In the composition at the abstract level we then reach $(q_{a_1}, q_{a_2}) \rightarrow$. Having $q'_{c_1} \precsim_{DS} q_{a_1}$ and $q_{c_2} \precsim_{DS} q_{a_2}$ then by our definition we reach $((q_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2}), (q'_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2})) \in R$ and R verifies the third clause of $\precsim_{DS}$.

   (b) Right τ : this is the symmetrical case of the Left τ.

4. **Delay :** λ is a time label $\lambda = \delta \in \Delta$. Since the advance of time is always synchronous, only the $SYNCHRONOUS$ composition is investigated. This leads to the case of the synchronous visible events, except for the use of the rule *Delay* instead of the rule *Visible Events*

of the timed weak simulation definition. We may finally conclude that $((q_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2}), (q'_{c_1}, q'_{c_2}), (q'_{a_1}, q'_{a_2})) \in R$ and R verifies the fourth clause of $\precsim_{DS}$. Finally, by following a similar reasoning as in the synchronous case, we show that (q'_{a_1}, q'_{a_2}) is coherent.

5. **Deadlock :** Suppose the transition $t_a : (q_{a_1}, q_{a_2}) \xrightarrow{\alpha} (q'_{a_1}, q_{a_2})$ is present in the valued composition (A, v_a) we have to prove that a corresponding transition exists in (C, v_c). For this, we investigate the three following cases :

   - Synchronous Case : since t_a exists on (A, v_a) then based on the $SYNCHRONOUS$ rule of the composition, $t_{a_1} : q_{a_1} \xrightarrow{\alpha} q'_{a_1}$ and $t_{a_2} : q_{a_2} \xrightarrow{\alpha} q'_{a_2}$ also exist at each side of the composition A_1 and A_2. Having by hypothesis $q_{c_1} \precsim_{DS} q_{a_1}$ and $q_{c_2} \precsim_{DS} q_{a_2}$ then by applying the rule *Deadlock* of $\precsim_{DS}$ on each of t_{a_1} and t_{a_2} we conclude respectively that there exists q'_{c_1} such that $t_{c_1} : q_{c_1} \xrightarrow{\alpha} q'_{c_1}$ and that there exists q'_{c_2} such that $t_{c_2} : q_{c_2} \xrightarrow{\alpha} q'_{c_2}$ with $v_{a_1}(q'_{a_1}) = v_{c_1}(q'_{c_1})$ and $v_{a_2}(q'_{a_2}) = v_{c_2}(q'_{c_2})$.
     Now, the composition of t_{c_1} and t_{c_2} would lead to $t_c : (q_{c_1}, q_{c_2}) \xrightarrow{\alpha} (q'_{c_1}, q'_{c_2})$. The target state is coherent and has the same valuation as its corresponding abstract state.
   - Left/Right Interleaving Cases : Similar proof as the previous Synchronous case.

We have shown that R satisfies the $\precsim_{DS}$ clauses. Thus the largest relation contains R. Since R contains the initial state couples $((q_{c_1}, q_{c_2}), (q_{a_1}, q_{a_2}))$, we can conclude that the largest relation contains it also. Consequently, the refinement property is satisfied by the two products.

□

3.4 Timed Weak Simulation Verification

We start by introducing our technique in the untimed context by giving a weak simulation μ-calculus-based property. This property is later extended to the timed context.

3.4.1 Composing the Abstract/Concrete Systems

Our technique shares its grounds and features with model-checking techniques. Indeed, the first step consists in composing asynchronously the abstract with the concrete system. Given an abstract CTTS $A = \langle Q_a, Q_a^0, T_a, L_\tau, \rho_a, \lambda_a, \iota_a, \triangleright_a \rangle$ and a concrete one $C = \langle Q_c, Q_c^0, T_c, L_\tau, \rho_c, \lambda_c, \iota_c, \triangleright_c \rangle$, the two systems are composed after having renamed the events of the two systems by indexing the abstract(resp. concrete) ones by a (resp. c). The composition is thus made asynchronous (Fig. 3.12) in order to be able to observe all the transitions of the concrete system and verify whether they are simulated by the abstract system or not. The synchronous composition is not applicable because transitions of

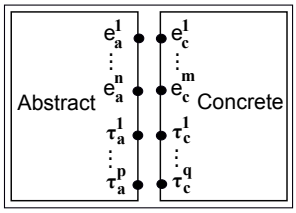


Figure 3.12 – *Systems*

the concrete system may disappear in the product system. This is the case when a concrete transition is not matched by the abstract system.

Untimed Weak Simulation Verification

The composition result is analyzed to check the weak simulation between A and C. To do so, the following *Weak Simulation* criterion [54, 28] which corresponds intuitively to the first two rules of the relation $\precsim$ is verified on $A \underset{\emptyset}{\|} C$:

$$\forall (q_a^0, q_c^0) \in Q_a^0 \times Q_c^0, (q_a^0, q_c^0) \models \nu X. \overbrace{\bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i \rangle \mathrm{X})}^{1} \wedge \overbrace{\bigwedge_j [\tau_c^j] X}^{2}$$

1. The first part of the formula means that for each concrete event e_c^i and for each transition labeled by this concrete event e_c^i, there exists a path of a number of abstract local events τ_a that leads eventually to a transition labeled by the abstract event e_a^i such that the target verifies recursively the property.
2. The second part of the formula means that after each transition labeled with a concrete local event τ_c^j the simulation is maintained.

Example of Untimed Weak Simulation Verification

We illustrate the intuition of the technique on an example of vending machine models [28].

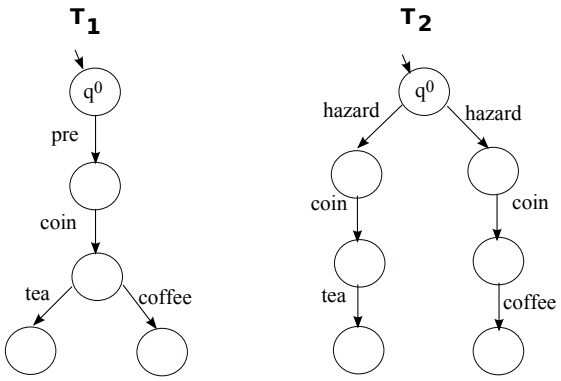


Figure 3.13 – *Example of Untimed Simulation Verification*

The following property is satisfied since from the initial state, for any transition selected in T_2, it is possible to select a transition in T_1 labeled with the same event. We then conclude that T_2 simulates T_1

$$(q^0_{T1}, q^0_{T2}) \models$$
$$\nu X.([coin_{T2}](\mu Y.(\langle pre\rangle Y \vee \langle coin_{T1}\rangle X))$$
$$\wedge[coffee_{T2}](\mu Y.(\langle pre\rangle Y \vee \langle coffee_{T1}\rangle X))$$
$$\wedge[tea_{T2}](\mu Y.(\langle pre\rangle Y \vee \langle tea_{T1}\rangle X)) \wedge [hazard]X)$$

However, checking the following property will return false meaning that T_1 does not simulate T_2. This is because after the coin event in T_1, it is possible to select either the coffee or the tea events, whereas in T_2, the coin transition leads to a state where only one of those events may be selected.

$$(q^0_{T1}, q^0_{T2}) \not\models$$
$$\nu X.([coin_{T1}](\mu Y.(\langle hazard\rangle Y \vee \langle coin_{T2}\rangle X))$$
$$\wedge[coffee_{T1}](\mu Y.(\langle hazard\rangle Y \vee \langle coffee_{T2}\rangle X))$$
$$\wedge[tea_{T1}](\mu Y.(\langle hazard\rangle Y \vee \langle tea_{T2}\rangle X)) \wedge [pre]X)$$

3.4.2 Extension to the Timed Context

We have seen a rather simple property that checks the weak simulation between two untimed systems via an asynchronous composition. However, this property could not be used directly in the timed context since it assumes that concrete and abstract events are composed asynchronously, while the composition of time transitions is necessarily synchronous because time advances at the same rate at the two sides of the composition.

Two alternatives are possible. The first is to specify these timing constraints in a timed variant of μ-calculus (for instance: [100]). This assumes the existence of a model-checking tool that supports such logic. The second is to specify the timing aspects with timed observers, compose the analyzed system with these observers and then make use of an untimed logic. We would rather follow the second technique.

For this purpose, we define two observers (Fig. 3.4 and 3.7) which are composed with both the concrete and abstract systems.

1. The first observer `ControlObserver` consists in observing the control aspects of the two systems. Namely, for each event of the implementation, the control observer will try to find a matching event of the abstraction that can happen at the same time.

2. The second observer `TimeObserver` models the elapse of time in the two systems.

In the two observers, the reset relations are empty. This way, the observers never impact the reset relations defined in the abstract and the concrete systems.

Control Observer

The Control Observer is depicted in Fig. 3.4.

At the initial state `ok`, the observer synchronizes with either one of the events e^i_a, τ^i_c or e^i_c. When synchronizing with any of the events e^a_i the observer signals an error (`err` state) since a concrete event is not yet found. When synchronizing with any of τ^i_c the observer maintains the state `ok`. Finally, when synchronizing with the concrete events e^i_c, it tries to match

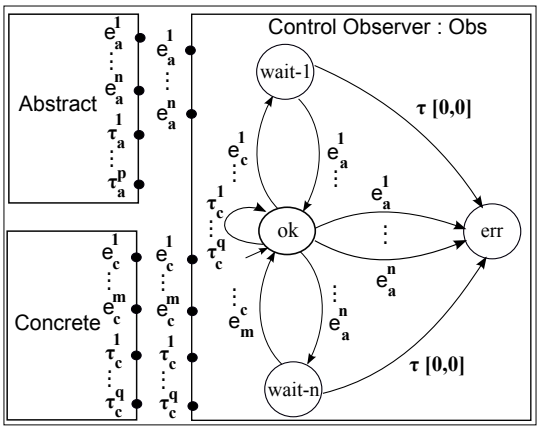


Figure 3.14 – *Control Observer*

them with the abstract events e_a^i. After a concrete event e_c^i is received, the observer transits to the state $wait_i$ meaning that it now awaits for a matching abstract event e_a^i. At this point, the following scenarios may happen :

- A matching abstract event is found and the observer transits back to the state `ok`.
- The abstract system violates the timing of the concrete system and the observer transits to the state `err`. That is, a matching abstract event is not possible at the same time as the corresponding concrete event. This is modeled by signaling an error in 0 u.t.. Hence, in case a matching event is found in 0 u.t, we would reach a non-deterministic choice between transiting back to `ok` or also to `err`. The two transitions would then be present in the composition process. As we will see, this choice is resolved in the μ-calculus property by searching for a path that satisfies the simulation and thus ignoring the error transition.

Time Observer

The control observer only checks whether two corresponding events could happen at the same time. However, it does not say anything about when an elapse of time has occurred. This leads to the definition of an additional observer `TimeObserver` (Fig. 3.7) in which two aspects are modeled. First, at the initial state `evt0`, only the transitions that are firable in 0 time can occur. This is done by specifying a concurrent choice between a timed event τ_0 constrained with $[0, 0]$ at one hand and all the events of the abstract and concrete systems at the other.

Second it makes visible the implicit elapse of time. At the state `Dly`, on each elapse of time, a timed event `delay` associated with the constraint $]0, \infty[$ is signaled. This event is later used by the μ-calculus property as a marker of an elapse of time.

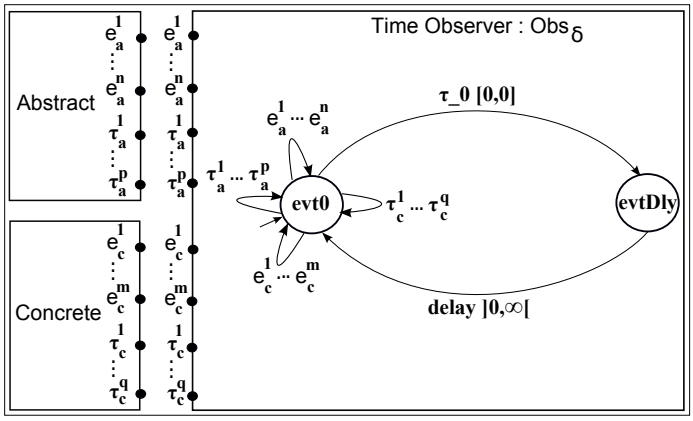


Figure 3.15 – *Time Observer*

Timed Weak Simulation Verification

Before giving the μ-calculus property used to verify the timed weak simulation, we make the following assumptions on the abstract and the concrete systems which are needed for verification purposes.

Assumption 1 (Concrete/Abstract) *A concrete system is any upper bounded closed CTTS. An abstract system is any τ non-Zeno, τ divergent, 1-τ, upper bounded closed CTTS.*

The CTTS hypothesis 1-τ is a sufficient condition for the following $\tau\delta$ permutation property.

Definition 3.24 ($\tau - \delta$ Permutation) *Given a TTS, for all $q\ q'\ q_1\ q_2 \in Q, \delta \in \Delta$, $q \xrightarrow{\delta} q' \wedge q \xrightarrow{\tau} q_1 \xrightarrow{\delta} q_2 \Rightarrow \exists q'', q' \xrightarrow{\tau} q'' \wedge q_2 \sim q''$.*

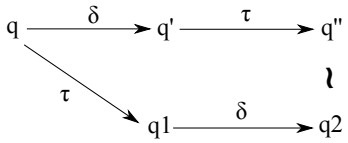


Figure 3.16 – $\tau - \delta$ *Permutation*

This means that from a state q, transitions τ and δ may be exchanged leading to similar states q_2 and q''. This property is close to the persistency of [87] that says that time cannot suppress the ability to do an action. However, our requirement that $q_2 \sim q''$ is stronger. This property is not true in general since the clocks newly reset at the state q'' have different values at q_2. The 1-τ hypothesis is a sufficient condition on the CTTSs so that the clock differences at q_2 and q'' would not affect the overall execution of the system.

Theorem 3.5 (1-τ is a sufficient condition for $\tau - \delta$ permutation) *Given a CTTS $\mathcal{T}$ that satisfies the 1-τ condition, its semantics $[\![\mathcal{T}]\!]$ verifies the property of permutation.*

Proof. Let $\mathcal{T} = \langle Q, Q^0, T, L_\tau, \rho, \lambda, \iota, \triangleright \rangle$ be a CTTS that satisfies the 1-τ condition and let $[\![\mathcal{T}]\!]$ be its semantics. We suppose that $\forall q\ q'\ q_1\ q_2 \in Q, \delta \in$

$\Delta, q \xrightarrow{\delta} q' \wedge q \xrightarrow{\tau} q_1 \xrightarrow{\delta} q_2$ (1) is verified in $[\![\mathcal{T}]\!]$. We prove that there exists a state q'' such that $q' \xrightarrow{\tau} q'' \wedge q_2 \sim q''$ (2). Actually in $[\![\mathcal{T}]\!]$ (1) is written as : $(q,v) \rightarrow^{\delta}_{[\![\mathcal{T}]\!]} (q, v+\delta) \wedge (q,v) \rightarrow^{\tau}_{[\![\mathcal{T}]\!]} (q',v') \rightarrow^{\delta}_{[\![\mathcal{T}]\!]} (q', v'+\delta)$ while (2) is written as $(q, v+\delta) \rightarrow^{\tau}_{[\![\mathcal{T}]\!]} (q', v'') \wedge (q', v'+\delta) \sim (q', v'')$. First, we investigate whether we can find $(q, v+\delta) \rightarrow^{\tau}_{[\![\mathcal{T}]\!]} (q', v'')$.

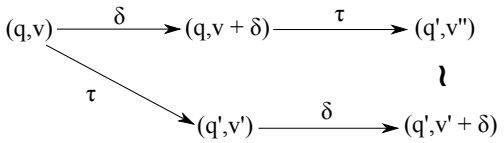


Figure 3.17 – *Sufficient Condition for $\tau - \delta$ Permutation*

Based on the CTTS semantics, we know the following, namely that $t : q \xrightarrow{\tau}_T q'$ exists:

$$\left(\begin{array}{c} \forall t', q \in \mathbf{dom}(\rho(t')) \Rightarrow v(t') \in \iota(t') \\ t : q \xrightarrow{\tau}_T q' \rightarrow, \quad \forall t', v'(t') = \begin{cases} 0 & \textbf{if } (q,q') \hookrightarrow t' \vee t \triangleright t' \\ v(t') & \textbf{else} \end{cases} , \\ q \in \mathbf{dom}(\rho(t)) \Rightarrow v(t) \in \overleftarrow{\iota(t)} \wedge v(t) + \delta \in \overleftarrow{\iota(t)} \\ \hline (q,v) \xrightarrow{\delta}_{[\![\mathcal{T}]\!]} (q, v+\delta) \wedge (q,v) \xrightarrow{\tau}_{[\![\mathcal{T}]\!]} (q',v') \xrightarrow{\delta}_{[\![\mathcal{T}]\!]} (q', v'+\delta) \end{array} \right)$$

Since $t : q \xrightarrow{\tau}_T q'$ and $(q,v) \xrightarrow{\delta}_{[\![\mathcal{T}]\!]} (q, v+\delta)$ exist, it means that at the CTTS level, τ is associated to an interval having a maximal bound greater or equal to δ. Then at the q state, τ may be delayed and is still a firable transition after δ. This also means that the enablement condition of τ depends only on q and not on the clock evolution. Thus, $(q, v+\delta) \rightarrow^{\tau}_{[\![\mathcal{T}]\!]} (q', v'')$ also exists.

Now in q', because of the $1 - \tau$ hypothesis, no new τ events are enabled. Thus, the $(q', v'+\delta)$ and (q', v'') are bisimilar thanks to the bisimilar states property 3.3.

□

Timed Weak Simulation Criterion The check of timed weak simulation consists in a μ-calculus property verified on the composition result of the abstract system, the concrete system and the two observers $(A \;|||\; C) \| (Obs \underset{L_\tau - A_r}{\|} Obs_\delta)$ where Obs is the control observer, Obs_δ is the time observer and $A_r = \{delay, \tau\_0\}$. The simulation of time transitions consists in verifying whether each delay made by the concrete system can also be made by the abstract one. But unlike the asynchronous composition in the untimed context with which we were able to alternate between the occurrence of the concrete and the abstract events, time is always synchronous in each of A, C and the two observers. The idea of alternating between the concrete and abstract systems does not apply to time transi-

tions. The *Timed Weak Simulation* criterion is defined as :

$$\forall (q_a^0, q_c^0) \in Q_a^0 \times Q_c^0, (q_a^0, q_c^0, ok, evt0) \models \nu X. \overbrace{Obs\ in\ ok \wedge Obs_\delta\ in\ evt0}^{(1)} \wedge \overbrace{\bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X) \wedge \bigwedge_j [\tau_c^j] X}^{(2)Weak\ Simulation} \wedge \overbrace{(\mathbf{EF}_{\tau_a}\langle delay\rangle \top) \Rightarrow \mathbf{EF}_{\tau_a}(\langle delay\rangle \top \wedge [delay] X)}^{(3)}$$

This property characterizes a set of product states to which the initial state must belong. This set of states is defined over the composition of states of A,C and the two observers. We comment on the *Timed Weak Simulation* criterion :

- (1) denotes the state in which concrete events are awaited.
- (2) is the untimed weak simulation criterion.
- (3) specifies that if time can elapse (delay event) in the product via a sequence of abstract local events -meaning that the time can elapse in the concrete system since the abstract system is τ divergent- then the time may elapse and for all possible delay events the simulation holds.

Remark 3.6 *We note here that the timed weak simulation criterion is correct without the hypothesis* $1 - \tau$*. However it is not complete. This is why we observe* τ *sequences. We discuss this in the following section.*

3.5 Soundness and Completeness of the Criterion

The proof of the correctness of the μ-calculus criterion w.r.t the mathematical definition of the timed weak simulation is based on the comparison between two relations defined as the largest relations which can also be seen as the greatest fixed points of monotone set functions.

In the following, we first present the adopted proof method. Then we apply it for the proof of the correctness of the timed weak simulation criterion.

3.5.1 Proof Method

In order to prove the inclusion between the greatest fixed points of two monotone set functions F_c and F_a in a modular way, we suppose that the two functions are defined as the intersection of k basic functions :

$$F_c(S) = \bigcap_{i=1}^{k} F_{c_i}(S) \quad , \quad F_a(S) = \bigcap_{i=1}^{k} F_{a_i}(S)$$

Then, we give the following sufficient condition that enables us to decompose the proof into k subproofs :

Definition 3.25 (Greatest Fixed Points Inclusion Criterion) *Given two functions* F_c *and* $F_a \in 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$*, we define the inclusion criterion as :*

$$F_c \sqsubseteq F_a \equiv \forall S_c S_a, S_c \subseteq S_a \cap F_c(S_c) \Rightarrow S_c \subseteq F_a(S_a)$$

Theorem 3.6 (Sufficient Condition for the Inclusion of Greatest Fixed Points) *Given a set $\mathcal{S}$ and two monotone functions F_c and $F_a \in 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ such that $F_\alpha(S) = \bigcap_{i=1}^{k} F_{\alpha i}(S)$, we have the following implication :*

$$(\forall i \in 1..k, F_{c_i} \sqsubseteq F_{a_i}) \Rightarrow max\ F_c \subseteq max\ F_a$$

Proof of the Sufficient Condition. We have $\forall S_c S_a i, S_c \subseteq F_{c_i}(S_c) \wedge S_c \subseteq S_a \Rightarrow S_c \subseteq F_{a_i}(S_a)$ and need to prove that $max\ F_c \subseteq max\ F_a$ with $max\ F$ is defined as $\bigcap_n F^n(\mathcal{S})$ [32]. We instantiate the hypothesis with $S_c = max\ F_c$. As $max\ F_c \subseteq F_c(max\ F_c)$, so $\forall i \in 1..k, max\ F_c \subseteq F_{c_i}(max\ F_c)$. Now, for $i \in 1..k$ we prove by induction on n that $\forall n, max\ F_c \subseteq F_{a_i}^n(\mathcal{S})$.

1. $n = 0 : max\ F_c \subseteq \mathcal{S}$.
2. Suppose $max\ F_c \subseteq F_{c_i}^n(\mathcal{S})$. By our sufficient condition we get $max\ F_c \subseteq F_{a_i}^{n+1}(\mathcal{S})$.

Thus $\forall i \in 1..k, max\ F_c \subseteq \bigcap_n F_{a_i}^n(\mathcal{S})$. We conclude that $max\ F_c \subseteq max\ F_a$. □

Definition 3.26 (Strengthened Greatest Fixed Points Inclusion Criterion) *Given a set $\mathcal{S}$ and two functions F_c and $F_a \in 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$, and a property $\mathcal{P}$ over $2^{\mathcal{S}}$ we define the inclusion criterion as :*

$$F_c \sqsubseteq_{\mathcal{P}} F_a \equiv \forall S_c S_a, \mathcal{P}(S_c) \wedge S_c \subseteq S_a \cap F_c(S_c) \Rightarrow S_c \subseteq F_a(S_a)$$

Theorem 3.7 (Strengthened Sufficient Condition for the Inclusion of Greatest Fixed Points) *Given a set $\mathcal{S}$ and two monotone functions F_c and $F_a \in 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ such that $F_\alpha(S) = \bigcap_{i=1}^{k} F_{\alpha\ i}(S)$, we have the following implication :*

$$(\forall i \in 1..k, F_{c_i} \sqsubseteq_{\mathcal{P}} F_{a_i}) \wedge \mathcal{P}(max\ F_c) \Rightarrow max\ F_c \subseteq max\ F_a$$

Proof of the Strengthened Sufficient Condition. The previous result cannot be reused but the proof is similar. □

The following proofs are made at the TTS level : we consider the semantics of the composition $(A \ ||| \ C) \| (Obs \underset{L_\tau - A_r}{\|} Obs_\delta)$ where $A_r = \{delay, \tau\_0\}$.

3.5.2 Introduction of Auxiliary Set Functions

The soundness and completeness of the μ-calculus criterion consists in showing its equivalence with the mathematical definition of the timed weak simulation. But while the mathematical definition is defined on two independent concrete and abstract systems, the criterion is defined on a time-synchronized composition. In order to allow their comparison, we simplify the μ-calculus property. Since the μ-calculus property characterizes a set of accepted states of the composition of the abstract and concrete systems with the two observers-for which the states are fixed (ok

and evto)-, this set of states may be written as the largest relation $\precsim_\mu$ such that :

$$
\begin{array}{l}
\forall q_1\ q_2, q_1 \precsim_\mu q_2 \Rightarrow \\
(\forall q_1'\ q_2'\ o\ e_c, (q_1, q_2, ok, evt0) \xrightarrow{e_c} (q_1', q_2', o, evt0) \Rightarrow \\
\quad \exists q_1''\ q_2''\ e_a, (q_1', q_2', o, evt0) \xrightarrow{\tau_a^* e_a} (q_1'', q_2'', ok, evt0) \wedge q_1'' \precsim_\mu q_2'') \wedge \\
(\forall q_1'\ q_2'\ o, (q_1, q_2, ok, evt0) \xrightarrow{\tau_c} (q_1', q_2', o, evt0) \Rightarrow o = ok \wedge q_1' \precsim_\mu q_2') \wedge \\
(\exists \delta > 0, (q_1, q_2, ok, o_\delta) \xrightarrow{\tau_a^* \delta} -) \Rightarrow \\
\quad \exists q_1'\ q_2'\ o\ (q_1, q_2, ok, evt0) \xrightarrow{\tau_a^*} (q_1', q_2', o, evtDly) \wedge \\
\qquad (\exists \delta > 0, (q_1', q_2', o, evtDly) \xrightarrow{\delta} -) \wedge \\
\qquad (\forall \delta > 0\ q_1''\ q_2''\ o', (q_1', q_2', o, evtDly) \xrightarrow{\delta} (q_1'', q_2'', o', evt0) \wedge q_1'' \precsim_\mu q_2'')
\end{array}
$$

Based on the definition of the CTTS composition and of the two observers, then by hiding the unused details of the composition, this property may be simplified to :

$$
\begin{array}{l}
\forall q_1\ q_2, q_1 \precsim_\mu q_2 \Rightarrow \\
\overbrace{(\forall q_1'\ e_c, q_1 \xrightarrow{e_c} q_1' \Rightarrow \exists q_2'\ e_a, q_2 \xrightarrow{\tau_a^* e_a} q_2' \wedge q_1' \precsim_\mu q_2')}^{(a)} \wedge \\
\overbrace{(\forall q_1', q_1 \xrightarrow{\tau_c} q_1' \Rightarrow q_1' \precsim_\mu q_2)}^{(b)} \wedge \\
\overbrace{\begin{array}{l}
(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2 \xrightarrow{\tau_a^* \delta} -) \Rightarrow \\
\quad \exists q_2', \quad q_2 \xrightarrow{\tau_a^*} q_2' \wedge \\
\qquad (\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta} -) \wedge \\
\qquad (\forall \delta > 0\ q_1'\ q_2', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow q_1' \precsim_\mu q_2'')
\end{array}}^{(c)}
\end{array}
$$

We now define $\precsim_\mu$ as the greatest fixed point of the monotone function F_μ defined over $Q_c \times Q_a$ where $F_\mu(S) = F_{\mu E}(S) \cap F_{\mu T}(S) \cap F_{\mu D}(S)$ with

$$
\begin{array}{l}
F_{\mu E}(S) = \{(q_1, q_2) \mid \forall q_1'\ e_c, q_1 \xrightarrow{e_c} q_1' \Rightarrow \exists q_2'\ e_a, q_2 \xrightarrow{\tau_a^* e_a} q_2' \wedge (q_1', q_2') \in S\} \\
F_{\mu T}(S) = \{(q_1, q_2) \mid \forall q_1', q_1 \xrightarrow{\tau_c} q_1' \Rightarrow (q_1', q_2) \in S\} \\
F_{\mu D}(S) = \{(q_1, q_2) \mid \quad (\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2 \xrightarrow{\tau_a^* \delta} -) \Rightarrow \\
\qquad\qquad \exists q_2', \quad q_2 \xrightarrow{\tau_a^*} q_2' \wedge \\
\qquad\qquad\quad (\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta} -) \wedge \\
\qquad\qquad\quad (\forall \delta > 0\ q_1'\ q_2', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow (q_1', q_2'') \in S)\}
\end{array}
$$

We give in the following the definition of $\precsim$ already seen earlier in this chapter. $\precsim$ is the largest relation such that :

$$
\begin{array}{l}
\forall q_1\ q_2, q_1 \precsim q_2 \Rightarrow \\
\\
(\forall q_1'\ e, q_1 \xrightarrow{e} q_1' \Rightarrow \exists q_2'\ e, q_2 \xrightarrow{\tau^* e} q_2' \wedge q_1' \precsim q_2') \wedge \\
(\forall q_1', q_1 \xrightarrow{\tau} q_1' \Rightarrow q_1' \precsim q_2) \wedge \\
(\forall q_1'\ \delta, q_1 \xrightarrow{\delta} q_1' \Rightarrow \exists q_2', q_2 \xRightarrow{\delta^*} q_2' \wedge q_1' \precsim q_2')
\end{array}
$$

Again we define $\precsim$ as the greatest fixed point of the monotone function F defined over $Q_c \times Q_a$ where $F(S) = F_E(S) \cap F_T(S) \cap F_D(S)$ with

$$F_E(S) = \{(q_1, q_2) \mid \forall q_1'\ e, q_1 \xrightarrow{e} q_1' \Rightarrow \exists q_2'\ e, q_2 \xrightarrow{\tau^* e} q_2' \wedge (q_1', q_2') \in S\}$$
$$F_T(S) = \{(q_1, q_2) \mid \forall q_1', q_1 \xrightarrow{\tau} q_1' \Rightarrow (q_1', q_2) \in S\}$$
$$F_D(S) = \{(q_1, q_2) \mid \forall q_1'\ \delta, q_1 \xrightarrow{\delta} q_1' \Rightarrow \exists q_2', q_2 \xRightarrow{\delta^*} q_2' \wedge (q_1', q_2'') \in S)\}$$

3.5.3 Proof of the Criterion

The proof consists in a soundness and a completeness proof of the criterion.

Soundness

Lemma 3.5 (Visible Events) *Given two CTTS A and C, we have :*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_c \subseteq S_a \cap F_E(S_c) \Rightarrow S_c \subseteq F_{\mu E}(S_a)$$

meaning

$$\forall S_c \subseteq S_a, \forall (q_1, q_2) \in S_c,$$
$$\overbrace{(\forall q_1'\ e_c, q_1 \xrightarrow{e_c} q_1' \Rightarrow \exists q_2'\ e_a, q_2 \xrightarrow{\tau_a^* e_a} q_2' \wedge (q_1', q_2') \in S_c)}^{(H)}$$
$$\Rightarrow (\forall q_1'\ e, q_1 \xrightarrow{e} q_1' \Rightarrow \overbrace{\exists q_2'\ e, q_2 \xrightarrow{\tau^* e} q_2' \wedge (q_1', q_2') \in S_a)}^{(G)}$$

Proof. Let $S_c \subseteq S_a$ and $(q_1, q_2) \in S_c$. Let $q_1' e$ such that $q_1 \xrightarrow{e} q_1'$. By the hypothesis (H), we obtain q_2' and e_a such that $q_2 \xrightarrow{\tau_a^* e_a} q'2 \wedge (q_1', q_2') \in S_c$. We choose them to be the witnesses for (G). As $S_c \subseteq S_a$ we get $(q_1', q_2') \in S_a$. □

Lemma 3.6 (τ Events) *Given two CTTS A and C, we have :*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_c \subseteq S_a \cap F_T(S_c) \Rightarrow S_c \subseteq F_{\mu T}(S_a)$$

Proof. This is similar to the previous proof. □

Lemma 3.7 (Delay) *Given two CTTS A and C supposed to be upper bounded closed and such that A is τ non-Zeno and τ-divergent :*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_c \subseteq S_a \cap F_D(S_c) \Rightarrow S_c \subseteq F_{\mu D}(S_a)$$

meaning

$$\forall S_c \subseteq S_a, \forall (q_1, q_2) \in S_c,$$
$$\overbrace{(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2 \xrightarrow{\tau_a^* \delta} -) \Rightarrow}^{(H)}$$
$$\exists q_2', \quad q_2 \xrightarrow{\tau_a^*} q_2' \wedge$$
$$(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta} -) \wedge$$
$$(\forall \delta > 0\ q_1'\ q_2'', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow (q_1', q_2'') \in S_c)$$
$$\Rightarrow (\forall q_1'\ \delta, q_1 \xrightarrow{\delta} q_1' \Rightarrow \overbrace{\exists q_2', q_2 \xRightarrow{\delta^*} q_2' \wedge (q_1', q_2') \in S_a)}^{(G)}$$

Proof. Let $S_c \subseteq S_a$.Let $(q_1, q_2) \in S_c$ such that (H). Let q_1', δ such that $q_1 \xrightarrow{\delta} q_1'$. We have to prove (G) :

1. if $\delta = 0$: let $q_2' = q_2$. We need to prove $(q_1, q_2) \in S_a$. But this is true thanks to $S_c \subseteq S_a$.
2. if $\delta \neq 0$: using the τ non-Zeno induction principle, we suppose that there exists a product state that is reached after $\delta_1 < \delta$ in S_c. Let st_0 be such a state. Since the abstract system is τ-divergent, then $q_2 \xrightarrow{\tau_a^* \delta}$ of left-side of the implication of (H) is satisfied. Thus there exists a state $x \in Q_c \times Q_a$ such that $st_0 \xrightarrow{\tau_a^*} x \xrightarrow{\delta > 0} -$ and $\forall \delta'$, there exists $x', x \xrightarrow{\delta'} x' \wedge x' \in S_c$. Three cases need to be investigated :
   (a) The delay in state x is in a right opened interval. Based on our hypothesis that the concrete and the abstract are upper bounded closed and on Property 1, the interval cannot be but unbounded. But this means that it is then possible to reach δ in the product. Let x′ be the associated product state. Thus $x' \in S_c$. Since $S_c \subseteq S_a$, we conclude that $x' \in S_a$.
   (b) The delay in state x is in a right closed interval with a bound greater or equal to δ. But this also means that it is possible to reach δ in the product. idem.

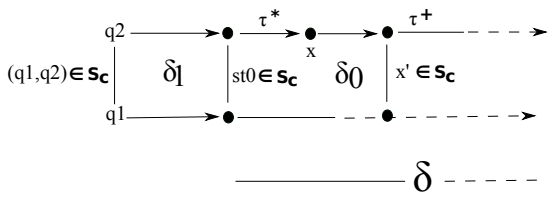


Figure 3.18 –

   (c) This corresponds to the case depicted in Fig 3.18. The delay in state x is in a right closed interval with a bound less than δ. This means that we reach the bound of the interval. Let δ_0 be this delay and $x \xrightarrow{\delta_0} x'$. In $x' \in S_c$, it remains to reach $\delta - \delta_1 - \delta_0 > 0$. But in the state x' of the product, only τ_a can occur because we have reached the bound of the interval. Using the τ non-Zeno

induction principle, the property that the abstract system can make a sequence $(\tau_a|\delta_i)$ such that $\Sigma\delta_i = \delta$ is verified.

□

Completeness

For the purpose of the completeness proof we introduce the following lemma.

Lemma 3.8 (Finiteness of Control) *Given a CTTS* $= \langle Q, Q^0, T, L_\tau, \rho, \lambda, \iota, \triangleright\rangle$ *and a monotonic property* $P_\delta \subseteq 2^Q$ *such that* $\forall\delta\delta', \delta \leq \delta' \Rightarrow P_\delta \subseteq P_{\delta'}$, *if the CTTS has a finite control (T is finite) then its semantics* $[\![CTTS]\!]$ *satisfies the following property allowing the quantifiers exchange :*

$$\frac{(\forall\delta > 0, \exists q', q \xrightarrow{\tau} q' \wedge P_\delta(q'))}{\exists q', q \xrightarrow{\tau} q' \wedge \forall\delta > 0, P_\delta(q')}$$

Proof. Given $q \in Q$, for any transitions t_1 and t_2 we say that $t_1 \leq t_2$ if $\{\delta \mid \exists q', (q, q') \in \rho(t_1) \wedge P_\delta(q')\} \subseteq \{\delta \mid \exists q', (q, q') \in \rho(t_2) \wedge P_\delta(q')\}$. $\leq$ is a total preorder because of the monotonicity of P. Since T is finite (by hypothesis) and non empty then there exists a greatest element. We choose its destination state. □

Lemma 3.9 (Visible Events) *Given two CTTS A and C, we have :*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_a \subseteq S_c \cap F_{\mu E}(S_a) \Rightarrow S_a \subseteq F_E(S_c)$$

Proof. This is similar to the Visible Events soundness proof. □

Lemma 3.10 (τ Events) *Given two CTTS A and C, we have :*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_a \subseteq S_c \cap F_{\mu T}(S_a) \Rightarrow S_a \subseteq F_T(S_c)$$

Proof. This is similar to the visible Events Soundness proof. □

For the purpose of the following proof, we use the Strengthened criterion by choosing for the property $\mathcal{P}(S)$ the stability of $S \subseteq Q_c \times Q_a$ under the strong bisimulation in Q_a. This means if $(q_c, q_a) \in S$ and $q_a \sim q'_a$ then $(q_c, q'_a) \in S$. Recall that stability is verified by *max F* because the strong simulation implies the weak timed simulation defined by *max F*.

Lemma 3.11 (Delay) *Given two CTTS A and C such that A has a finite control* [2] *and is 1-*τ *:*

$$\forall S_c S_a \in 2^{Q_c \times Q_a}, S_a \subseteq S_c \cap F_{\mu D}(S_a) \wedge S_a \text{ is } \sim -stable \Rightarrow S_a \subseteq F_D(S_c)$$

meaning

[2] This is not a restriction since the CTTSs are user defined and their set of transitions is always finite. The semantics of the CTTS may still be infinite due to time transitions.

$$\forall S_c \subseteq S_a, \forall (q_1, q_2) \in S_c,$$
$$\overbrace{(\forall q_1' \ \delta, q_1 \xrightarrow{\delta} q_1' \Rightarrow \exists q_2', q_2 \xRightarrow{\delta^*} q_2' \wedge \Sigma\delta_i = \delta \wedge (q_1', q_2'') \in S_a)}^{(H)}$$
$$\Rightarrow (\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2 \xrightarrow{\tau_a^* \delta} -) \Rightarrow$$
$$\exists q_2', \quad q_2 \xrightarrow{\tau_a^*} q_2' \wedge$$
$$(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta} -) \wedge$$
$$(\forall \delta > 0 \ q_1' \ q_2'', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow (q_1', q_2'') \in S_c)$$

Proof. Let $S_a \subseteq S_c$. Let $(q_1, q_2) \in S_a$ such that (H). Let $\delta > 0$ such that $hyp_2 : q_1 \xrightarrow{\delta} - \wedge q_2 \xrightarrow{\tau_a^* \delta} -$. We need to prove that :

$$\left(\begin{array}{l} \exists q_2', q_2 \xrightarrow{\tau_a^*} q_2' \wedge (\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta}) \wedge \\ \qquad (\forall \delta > 0 \ q_1' \ q_2'', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow (q_1', q_2'') \in S_a) \end{array} \right) \tag{3.1}$$

Two cases need to be investigated which are whether from the state (q_1, q_2) a delay event may occur or not:

1. Suppose $hyp_3 : (q_1, q_2) \models \langle delay \rangle \top$: Let $q_2' = q_2$, so $q_2 \xrightarrow{\tau_a^*} q_2'$ is verified and by hyp_3, $(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta})$ is also verified. We are left with proving :

   $$(\forall \delta > 0 \ q_1' \ q_2'', q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2'' \Rightarrow (q_1', q_2'') \in S_a)$$

   Let $\delta > 0 \ q_1' \ q_2''$ such that $q_1 \xrightarrow{\delta} q_1' \wedge q_2' \xrightarrow{\delta} q_2''$. By hyp_1 we know that $\exists q_2', q_2 \xrightarrow{(\tau_a|\delta_i)^*} q_2' \wedge \Sigma\delta_i = \delta \wedge (q_1', q_2') \in E_2)$. Let q_2' such a state. And since by hypothesis $E_2 \subseteq E_1$ then $(q_1', q_2') \in E_1$. Now having $q_2 \xrightarrow{\delta} q_2''$ and $q_2 \xrightarrow{(\tau_a|\delta_i)^*} q_2'$ then by the permutation property we have $q_2' \sim q_2''$. By the stability of S_a for the strong bisimulation we obtain $(q_1', q_2'') \in E_1$. Since $S_a \subseteq S_c$, we have $(q_1', q_2'') \in E_1$.

2. Suppose $hyp_4 : (q_1, q_2) \not\models \langle delay \rangle \top$: By hyp_2 we know that there exists st' such that $q_2 \xrightarrow{\tau_a^*} st'$ and $(q_1, st') \xrightarrow{\delta} -$. Knowing that the abstract system is 1-τ then either $st' = st$ or $st \xrightarrow{\tau} st'$. But by hyp_4, the first case is impossible, then $st \xrightarrow{\tau} st'$.

   Let δ be a delay $\neq 0$ and q_1' such that $q_1 \xrightarrow{\delta} q_1'$. By hyp_1, the abstract system contains a path of $\tau - \delta$ of whatever duration made by the concrete system that preserves the refinement $(\in E_2)$. This path surely starts with a τ since time cannot elapse in (q_1, q_2). By the hypothesis of finiteness of control of the abstract system, it is possible to find a unique τ transition as a first step of all the paths.

   Let $q_2 \xrightarrow{\tau} x_0$ be this transition. We choose x_0 as the state q_2' of the property (1) we are proving. Three conditions have to be satisfied :

   (a) $q_2 \xrightarrow{\tau_a^*} x_0$: This is verified by construction of x_0.

(b) $(\exists \delta > 0, q_1 \xrightarrow{\delta} - \wedge q_2' \xrightarrow{\delta} -)$: This is also verified. We take δ already introduced. We have $q_1 \xrightarrow{\delta} q_1'$. We also have $x_0 \xrightarrow{\delta} -$ since another τ is no longer possible because of the 1-τ hypothesis.

(c) Let $\delta_0 > 0$ q_1' q_2'' such that $q_1 \xrightarrow{\delta_0} q_1' \wedge x_0 \xrightarrow{\delta_0} -$. By our choice of x_0, there exists a path of duration δ_0 starting from x_0 in the abstract system and preserving the refinement. Knowing that there is only one τ, this path does not contain but a unique step, the δ_0 delay. Let x_1 such that $x_0 \xrightarrow{\delta_0} x_1$. We have $(q_1', x_1) \in E_2 \subseteq E_1$.

□

Discussion On the Assumptions

We discuss the two major restrictions made on the abstract system :

1. τ non-Zenoness and τ divergence: these two are standard assumptions made on timed systems. In our context they guarantee the progress of time in the abstract system. This is necessary in our composition-based method because time is always synchronous. Blocking the time in the abstract system could thus result in blocking time in the whole composition and hide concrete delays.

2. No successive τ transitions : permitting τ transitions in A complicates the verification of the timed weak simulation, because in this case any delay in C can be matched by a series of delays in A separated by τ transitions [36]. An example that illustrates this problem is depicted in 3.19. In the composition of the abstract and the concrete system, a trace where an event a_2 is not matched with an event a_1 exists. This means that the mu-calculus property will not be satisfied even though a simulation does hold. This belongs to the case where an elapse of 3 u.o.t is consumed entirely by the first time interval of the abstract system $[0, 3]$. The occurrence of a_2 would thus never be followed instantaneously by a matching a_1 event.

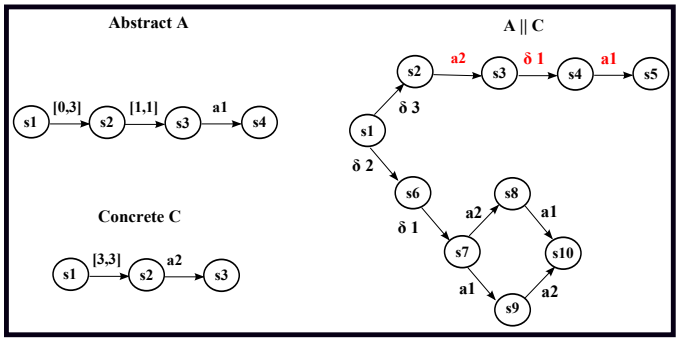


Figure 3.19 – *Counter Example*

Now concerning our 1-τ restriction, general modeling techniques of real time systems are still permitted. For instance, specifying a maximal bound of a global event e is made by a choice between a timed local event τ and the event e. Specifying a minimal bound of a global event e is made by a sequential execution of a timed local event τ and e.

3.5.4 Extension to a Deadlock-Sensitive Timed Weak Simulation

The check of Deadlock-Sensitive (DS) timed weak simulation consists also in a μ-calculus property verified on the composition result of the abstract system, the concrete system and the two observers $(A \;|||\; C) \| (Obs \underset{L_\tau - A_r}{\|} Obs_\delta)$ where Obs is the control observer, Obs_δ is the time observer and $A_r = \{delay, \tau\_0\}$. The *DS Timed Weak Simulation* criterion is written in μ-calculus as follows :

$$\overbrace{\forall (q_a^0, q_c^0) \in Q_a^0 \times Q_c^0, (q_a^0, q_c^0, ok, evt0) \models}^{TimedWeakSimulation}$$
$$\overbrace{\nu X.Obs \text{ in } ok \wedge Obs_\delta \text{ in } evt0 \wedge \bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X) \wedge \bigwedge_j [\tau_c^j] X \wedge}$$
$$(\mathbf{EF}_{\tau_a}\langle delay\rangle \top) \Rightarrow \mathbf{EF}_{\tau_a}(\langle delay\rangle \top \wedge [delay] X) \wedge$$
$$\overbrace{\bigwedge_i [e_a^i]\langle e_c^i\rangle \top}^{Event\ Deadlock}$$

This property characterizes a set of product states to which the initial state must belong. This set of states is defined over the composition of states of A,C and the two observers. We comment on the *DS-Timed Weak Simulation* criterion :

- The first part is exactly the Timed Weak Simulation criterion.
- The second part describes the deadlock preservation property. Actually it describes that each abstract visible event is followed by a corresponding concrete event.

The proof of equivalence of this criterion with the DS-timed weak simulation is direct their formulations are similar.

3.5.5 Extension to a State-Event Timed Weak Simulation

The DS-timed weak simulation criterion is extended to the case of the state-event timed weak simulation with the addition of the relation between the concrete and the abstract propositions. For $\mathcal{P}$ the set of propositions and a proposition $p \in \mathcal{P}$, the μ-calculus property is then extended with this proposition relation as shown in the following :

$$\overbrace{\forall (q_a^0, q_c^0) \in Q_a^0 \times Q_c^0, (q_a^0, q_c^0, ok, evt0) \models}^{DSTimedWeakSimulation}$$
$$\overbrace{\nu X.Obs \text{ in } ok \wedge Obs_\delta \text{ in } evt0 \wedge \bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X) \wedge \bigwedge_j [\tau_c^j] X \wedge}$$
$$(\mathbf{EF}_{\tau_a}\langle delay\rangle \top) \Rightarrow \mathbf{EF}_{\tau_a}(\langle delay\rangle \top \wedge [delay]\mathbf{EF}_{\tau_a} X) \wedge$$
$$\overbrace{\bigwedge_i [e_a^i]\langle e_c^i\rangle (\bigwedge_{p \in \mathcal{P}} p_c \Leftrightarrow p_a) \wedge}^{State-Event\ Deadlock}$$
$$\overbrace{\bigwedge_{p \in \mathcal{P}} p_c \Leftrightarrow p_a}^{PropositionsRelation}$$

where $(q_c, q_a) \models p_c$ if $p \in v_c(q_c)$ and $(q_c, q_a) \models p_a$ if $p \in v_a(q_a)$. We comment on the addition of the propositions relation :

- We say that for each proposition p, if p is satisfied by the variables of the concrete system x_c^k, then the variables of the abstract system x_a^l should satisfy it as well. Obviously, the concrete and abstract variables would depend on the systems that are verified.

The proof of equivalence of this criterion with the State-Event DS-timed weak simulation is direct because their formulations are similar. We just say here that in the given formula, the propositions are verified at the initial state and then after the occurrence of the visible events. This is because the recursivity of the formula is applied after the event occurrence in $\bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X)$. This coincides with our mathematical definition of the simulation.

3.6 Conclusion

We have given in this chapter our first main contribution. This contribution consists in the study of timed simulation relations and their properties. For this we have defined a timed weak simulation that holds in both the state and the action based contexts. We have shown that in order for the weak simulation to preserve linear logic properties, other clauses namely the deadlock and divergence need to be verified. We have also given a verification technique for the simulation. In the next chapter, we tackle the application part of this document. For this, we start by introducing the BPEL language and its relation with formal methods.

Part III

Application

Chapter One

Web Services

1.1 Introduction

With the emergence of distributed systems technologies and the fast evolution of service oriented architecture, Web services are increasingly becoming a major part of our daily lives. Many web services composition languages have been developed to describe the way a group of distributed web services interact with each other. In this matter, the Web Services Business Process Execution Language (WS-BPEL or simply BPEL) [18] is a well known service composition language. However, BPEL lacks a formal semantics and is defined informally in natural language. Several transformations to formal formalisms have thus been suggested. Their common goal is the verification of BPEL processes and their composition.

This chapter consists of two main sections. In the first section we introduce the web services and their composition. Afterward, we present the BPEL, which is the web services composition language that will be used all along this document. In the second section, we present the relation between BPEL and formal methods by covering a subset of transformations from BPEL to formal languages.

1.2 Service Oriented Architecture

Service oriented architecture (SOA) is a form of a distributed architectural approach that allows to create independent systems and applications that communicate with each other by exposing and using services. In today's world, SOA is considered a key architectural pattern for prompt and rapid integration, leading thus to a more agile Information Technology (IT). Indeed, most of today's greatest successes, in terms of bringing agility to the whole enterprise through its IT backbone, have been provided by SOA and its major technological counterparts that are the Web Services and the Enterprise Service Bus (ESB) [98]. SOA is typically used as a communication way between enterprises and clients. An enterprise then offers its products in the form of a set of Web services publishing operations.

1.2.1 Web Services

The Web Services represent a communication mechanism between two distant applications over the web. The main characteristics of a web service are the following :

1. They use mostly the HTTP protocol as a communication protocol.
2. They are specified in an XML-based notation to describe the exchanged messages and data.
3. They organize the order of messages invocations and responses.

Web Services Layers

The web services architecture is based on a layered model that contains the following three main layers :

1. Messages : in this layer the structure of the exchanged messages is described. This is done using the Simple Object Access Protocol (SOAP) [3] that provides a messaging framework that allows the exchange of messages coming from different protocols such as HTTP, SMTP, FTP etc .
2. Descriptions : in this layer, the web services interface is described. This is done using the Web Service Description Language (WSDL) [84] which is an XML-based language that allows to describe the functionality offered by a Web service. A WSDL description of a web service provides the address of the web service (how it can be called) as well as the operations (input/output parameters in the form of messages) that the service offers.
3. Processes : in this layer, the dynamic behavior of a web service may be described, such as the order in which certain messages invocations and responses is made etc. This can be written using the Business Process Execution Language BPEL. In this layer also, the discovery of services is made which allows to search for a specific web service in the service registry of an enterprise.

In the context of this chapter we emphasize on two layers which are the description layer in which the WSDL is used and the processes layer in which the BPEL is used.

WSDL The Web Services Description Language WSDL [84] is an XML-based language that describes the interface and the functionality offered by a Web service. The web service interface defines the address of the service, the operations it offers, the operation's input and output parameters along with their data types. WSDL describes the functionality of a web service with a set of operations which define the type of exchanged messages. The operations are then encapsulated by an endpoint (or port) that is linked to a concrete transport protocol such as SOAP, HTTP etc which allows the transport of the messages between the web service and its clients. We show the structure of a WSDL file :

```
<definitions name=...>
        <types> Data Types used by the messages </types>
        <message name=...> Message Definition </message>
        <portType name=...>+ Set of operations
                <operation name=...> Operation Definition </operation>+
        </portType>+
        <binding name=...> messages link with the implementation </binding>
        <service name=...> Service and URL Definition </service>
</definitions>
```

A WSDL file starts with the root element `definitions`. It contains the name of the web service and may also reference other name-spaces that are used by the WSDL file. Inside the `definitions` the following elements are defined :

1. types : defines the data-types used by the messages. The data-types may be XML schema written in the XML Schema language [4] (also known as XSD).

2. message : defines the exchanged data independently of the implementation and the transport protocol. The message contains the information needed to perform the operation of the web service.

3. operation : describes the functionality offered by a web service. An operation defines the message exchange between a web service and its clients. An analogy may be made between an operation and a method or function call in a traditional programming language.

4. portType : a porttype encapsulates a set of operations. This porttype is then associated to a transport protocol and is used as an endpoint of the connection. A group of porttypes actually define the interface of a web service, its operations and the messages that may be performed by the operations.

5. binding : defines the real link between the operation messages and the transport protocol. It also defines the SOAP binding style (RPC/-Document) and transport (SOAP Protocol).

6. service : defines the access ports that attaches the bindings to the URL.

1.2.2 Web-Services Composition

Until now we have only seen the way a web service can be described, but did not talk about how it can be composed. Actually, a greater value results of a composition of multiple web services - leading to a new web service- each of them dedicated to the handling of one task in a business procedure. For instance, a merchandising company can decide to create two web services, one for invoice management and another for shipment preparation. The composition of these two will create a new web service describing the merchandising company. In the following we will describe the two most used approaches for web services composition.

Orchestration vs. Choreography

The terms orchestration and choreography [90] describe two web services composition approaches that allow to create new web services out of existing ones. Choreography is in nature a collaborative effort in which the message sequences are tracked between the interacted web services. Each web service describes the role it plays in the interaction by receiving or sending messages. In a choreography, each web service knows exactly when to do an interaction, the operations it needs to execute and the parties with which it interacts. In Fig 1.1 we show a choreography of four web services. The interaction is started by WS1 that invokes WS2. As WS2 turn comes up, it invokes WS3 and WS4 concurrently and then awaits for their sequential responses before returning a response back to WS1.

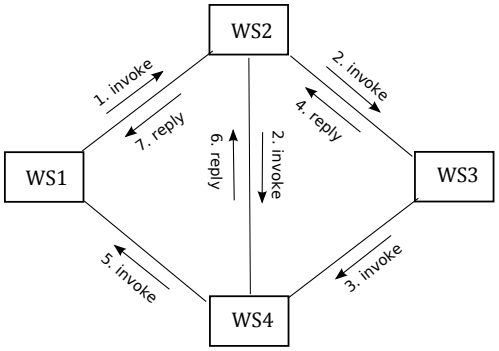


Figure 1.1 – *Choreography*

Unlike choreography, orchestration is an approach that is based on having a central process called orchestrator that controls the order and the way the different web services interact. This is done by means of business tasks or activities. The interaction is also made by receiving and sending messages but this time the interacting web services do not know they are involved in a business composition, only the orchestrator is aware of this business goal. So, in an orchestration the control is always represented from the orchestrator's perspective. In Fig 1.2, we show how SW2 plays the role of the orchestrator. First it receives the SW1's message, then it communicates concurrently with SW3 and SW4 before replying to SW1.

Business Process Execution Language BPEL

BPEL4WS (or just BPEL) [18] is an XML-based language that describes the behavior of web services in a business process interaction. The interaction of different partner web services is done for orchestration purposes. BPEL is defined over a WSDL file that describes the operations allowed by the BPEL process, while the BPEL process defines the order in which these operations are executed.

Structure of a BPEL process A BPEL process contains a primary activity -containing nested activities- that may communicate with the partner web services in order to allow their orchestration. Each web service consists of

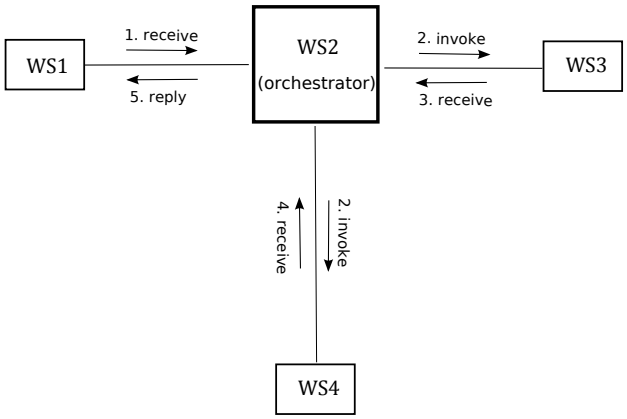


Figure 1.2 – *Orchestration*

a partnerlink, a portType and an operation that other web services may invoke in order to start a specific communication. The communication with the different partners is made by sending and receiving messages via this tuple. The partnerlink is a BPEL communication point. Depending on whether the communication is synchronous or asynchronous, the partnerlink contains one or two portTypes (see WSDL 1.2.1). The second portType is used to enable callBacks in asynchronous communications.

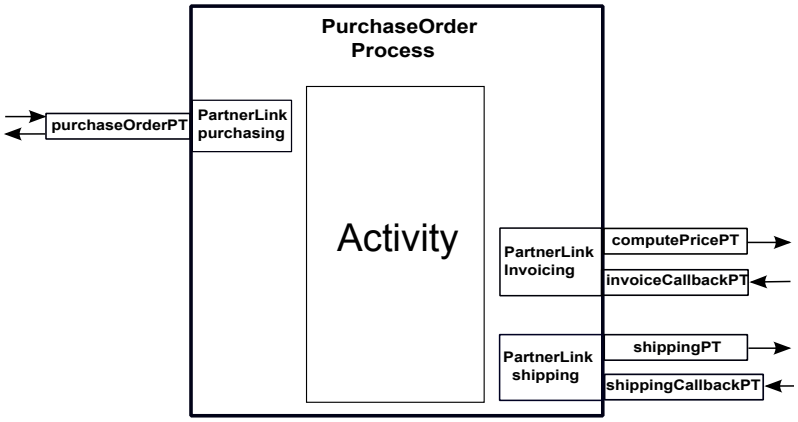


Figure 1.3 – *BPEL Structure: ex PurchaseOrder*

In Figure 1.3, we show the structure of the PurchaseOrder process [18] example. The process starts by receiving a purchase demand over the Partnerlink purchasing and the portType purchaseOrderPT. It then invokes two web-services (1) over the Partnerlink Invoicing and the PortType ComputePricePT in order to prepare the invoice on one hand and (2) over the Partnerlink shipping and the portType shippingPT to manage the shipping at the other hand. Once a reply from each of these two web-services is received respectively over the portTypes invoiceCallbackPT and shippingCallbackPT, the BPEL process sends back an invoice on the same partnerlink, the one that has received the purchase demand.

BPEL Language We show in the following a simplified structure of a BPEL process :

```
<process name="NCName" targetNamespace="anyURI"
...
<partnerLinks>?
<!-- Note: At least one role must be specified. -->
...
</partnerLinks>
<variables>?
...
</variables>
<faultHandlers>?
<!-- Note: There must be at least one faultHandler -->
...
</faultHandlers>
<eventHandlers>?
<!-- Note: There must be at least one onEvent or onAlarm. -->
...
</eventHandlers>
activity
</process>
```

1. *Partnerlinks* represent the static part of a BPEL process and are described in a WSDL document where the operations offered by a web service are also given. The dynamic part of BPEL is described by means of activities.

2. *Variables* are used to save the values of the input and output messages of the BPEL process. Their types may reference either a message type in a WSDL file or an XML schema data type.

3. *Basic Activities* define the elementary operations of the business process. They are the usual ones such as `Receive`, `Reply`, `Invoke`, `Assign`, `Throw or Rethrow`, `Exit`, `Empty`, `Wait` to delay the execution and the less usual ones such as `Compensate` and `CompensateScope` to trigger the compensation handlers.

4. *Structured Activities* define the order in which nested activities are executed. These are the `Sequence` and the `Flow` activities for the sequential and parallel execution respectively, the `While`, the `RepeatUntil`, the `If`, the `Pick` for a choice over message events or alarm events and the `ForEach`. Finally, the `Scope` activity to which one can attach the BPEL handlers may be used as a container of BPEL activities.

5. *Handlers :* BPEL provides a fault handler for internal faults handling, a compensation handler for undoing successfully finished scopes, a termination handler that controls the forced termination of a scope and an event handler that manages message and alarm events.

Moreover, links may be used inside a `Flow` activity to provide additional control over the order of execution of parallel activities. Each activity may have multiple outgoing links as well as multiple incoming links. Every outgoing link has an associated transition condition. The transition conditions are Boolean expressions based on the process data which are written in other languages such as XPath [27]. When an activity completes, it sets its outgoing links by evaluating their transitions conditions. Now before the start of activities, these transition conditions are evaluated by the targeted activities in the form of a joinCondion. If the joinCondition evaluates to true, the activity is started. Otherwise the activity is not

started and a standard bpel:joinFailure fault MUST be thrown, unless the value of suppressJoinFailure is yes in which case bpel:joinFailure is not thrown [18]. In this latter case (suppressJoinFailure = yes), the outgoing links of the activity MUST be assigned a false status. This approach is called Dead-Path Elimination (DPE) [18].

Timed Constructs use relative time within the `For Duration` primitive and absolute time within the `Until deadline` primitive. These features are used by a `wait` activity or an `<on alarm/>` event of a pick activity or of an event handler.

What's Behind the Language Here we give in a nutshell some of the interesting points of the BPEL language.

- BPEL is a hierarchical component-based language where structured activities are the nodes of the process and basic activities are its leaves.
- The concurrency is well integrated in BPEL, and concurrent BPEL activities may communicate either by shared variables (BPEL variables) or by valued events (links and attached Join Conditions).
- Concerning the BPEL assignments, the WS-BPEL 2.0 specification requires the *assign* activity to be atomic. That is, either all the copies succeed or no changes are made.

These points are important to know in the context of the transformation to FIACRE. As we have seen, FIACRE is also a component-based language where communications happens through shared variables or valued events. Moreover the transitions in FIACRE are also atomic.

1.3 Formal Methods for the Verification of Web Services Composition

Several works have been pursued in the area of modeling and verification of BPEL processes (see [33] for an overview). Most of this work [97, 47, 74] is based on using an intermediate formalism to model the activities of BPEL and then applying verification techniques on the obtained system. Usually, the obtained model is analyzed through the use of techniques and tools built around the target formalism. These proving techniques vary between automatic or semi-automatic ones. In the following we give a subset of these techniques which we sort based on the target formalism.

1.3.1 BPEL to Petri Nets

The first group of work was interested in mapping BPEL constructs to Petri Nets. The first to address an extensive mapping from earlier versions of BPEL to Petri Nets was the work of [60] and [101]. In this transformation the data are abstracted, along with timed constructs. This transformation supports the verification of behavioral properties such as the absence of deadlock. The obtained model is then used for this purpose. A tool, BPEL2PN, that embeds this transformation is also developed [60].

The work of [60] has been extended in [74] in order to cover the new features of BPEL 2.0. This transformation also abstracts from data and time modeling, but it covers all the features of BPEL 2.0. Lohman has also been interested about the composition of multiple BPEL processes [76] in order to verify the composition as a whole. A tool for this transformation also exists [75].

1.3.2 BPEL to process algebras

Mappings from BPEL to process algebras as CCS [83] and LOTOS [31] have been suggested. Indeed, in [97] the authors show how CCS can be exploited to treat BPEL constructs and to verify CTL properties. Moreover, a two-way mapping between BPEL and LOTOS is studied in [47]. This technique is quite interesting because it allows the conception of web services at the two levels. It also allows to reason about the equivalences between services by making use of the bisimulation notion.

Others have been interested in transformations towards Promela. In the work of [85], a subset of the BPEL constructs are transformed into Promela -via an intermediary formalism - and connected to other processes in the description. LTL properties are then verified on the result process by applying the Spin model checker [63] in order to detect deadlock in the execution traces [86]. [48], [49] and later [35] have studied the modeling of a choreography of multiple BPEL processes with guarded state-transition systems expressed in Promela. The Promela systems are then verified using the Spin model checker. In [48] , the data-types expressed in XML and XPATH are modeled using Promela. However, this is only used for the case of finite enumerated data-types because of state explosion. A tool (WSAT) [50] has been also developed.

1.3.3 BPEL to Timed Formalisms

Some works were interested in modeling the timed aspects of BPEL. We quote in this matter the work of [69] in which the timed behavior of BPEL activities are captured by introducing a formalism called Web Service Timed State Transition Systems (WSTTS) based on timed automata. In this work, timeouts and the absolute time of BPEL are treated. In addition, they consider in this technique the temporal cost of activities and deal with synchronous services. They support as well the expression of complex timed properties in Duration Calculus and verify them using NuSmv. However, in this technique, only a discrete notion of time is considered. Another work based on a transformation towards timed automata (TA) is also presented in [93]. An algorithm for mapping these patterns to timed automata is later integrated [92] in the `ActiveBPEL` tool. This work covers both the relative and the absolute time of BPEL. The compensation and the termination handlers are not modeled. Moreover, this transformation does not provide a way to explicitly specify the duration of synchronous calls or complex time-related properties.

In [99] the authors are interested in determining the execution time of BPEL activities by modeling the timed behavior of BPEL in timed Petri Nets. Nevertheless, the absolute time is treated by assuming that the current time is given explicitly.

1.3.4 BPEL to Event_B

Approaches based on proof methods have also been used namely through a mapping from BPEL to Event_B [8]. In this technique [10], the WSDL file is translated to an Event-B context and the body of BPEL process is translated to one or several Event_B machines. Unlike model checking based techniques, this technique does not abstract from data. Thus, data-based control is modeled. However, time is not modeled since it is not supported by Event-B. Apart of this, all the BPEL 2.0 constructs are taken into account. The advantage of this technique, is that it does not face the problem of state explosion encountered in model checking techniques. However, the proof obligations produced by the Event_B require an interactive assistance and are usually complex to achieve. Now, concerning the supported properties, the verification of all temporal logic properties is not explicit. This is a consequence of the target formalism Event_B. The verified properties are restricted to static properties such as type properties, to safety properties, deadlocks, simple liveness properties and to properties related to transactions.

1.3.5 Formal Methods and BPEL Summary

To conclude this chapter, we give in the following a summary of the existing BPEL verification tools.

Project	[97, 47][85]	[60, 74]	[10]	[69]	[93]
Formalisms	LOTOS/CCS/Spin	Petri Nets	Event B	WSTTS	TA
BPEL 2.0	−	+	+	+	−
Coverage	+	++	++	+	+
Logic	temporal,LTL	CTL	1st order	DC	CTL
Time Modeling	−	−	−	+	+
Timed Properties	−	−	−	+	−

Figure 1.4 – *Brief BPEL verification tools coverage*

1.4 Conclusion

We have introduced in this chapter the application part of this PhD. Notions such as web services, web services composition, and web services verification were discussed. We have shown that a strong relation exists between BPEL, a web service composition language, and formal methods. In the next chapter, we tackle the second contribution of this PhD which consists in a transformation from BPEL to FIACRE.

Chapter Two

Modeling BPEL in FIACRE

2.1 Introduction

Formal methods have been used to verify BPEL models and similar languages. This is done via a transformation from BPEL to several formal specification languages. The goal of this transformation is either to give formal semantics to languages that lack such feature (BPEL in our case), or to use the resulted formal code for verification purposes.

In this chapter we define a transformation from BPEL to the formal specification language FIACRE. The transformation covers a subset of the BPEL constructs which will later appear in the use case that illustrates our technique. Other constructs are given in the Appendix B. The interesting point about this transformation is that it is (1) well-structured, (2) proven to be semantically correct w.r.t some BPEL properties and (3) treats the timed aspects of BPEL.

- By well-structured we mean that the same composition and hierarchy of the BPEL activities are maintained in the transformation. This comes essentially from the fact that the structure of BPEL is natively present in FIACRE since both languages are component based ones. This allows a compositional transformation. Moreover, the parallelism and the communication notions of BPEL (shared variables and valued events) are close to the ones integrated in FIACRE. Finally, the atomicity of the BPEL assign activity is also maintained by FIACRE.
- We show how the transformation can be proved correct by checking BPEL semantic properties on the generated model.
- The timing aspects of BPEL are considered in this transformation. FIACRE natively supports real time constraints.

In the next Section we start by identifying some BPEL semantic properties we want to preserve by our transformation. Afterward, we give our transformation for a subset of BPEL constructs. Finally, we show how to prove that the FIACRE patterns preserve these BPEL semantic properties.

2.2 BPEL Semantics

Even though the semantic properties of BPEL are given in natural language, they are quite clear concerning some specific points. In the following, we collect some of these clauses that we would like to be verified by the FIACRE transformation. We note that these clauses are not exhaustive but are rather a subset of the BPEL constructs. This work may be extended to all of the BPEL constructs.

- Control Semantics : ignoring links, the semantics of the business processes, <scopes>, and structured activities determine when a given activity is ready to start. For example, the second activity in a <sequence> is ready to start as soon as the first activity completes. Similarly, an activity nested directly within a <flow> is ready to start when the <flow> itself starts (Links Semantics : page 105 of [18]).

- Links Semantics : when an activity completes without propagating any fault it triggers all of its outgoing link. When an activity is ready to start (part of the control flow) and the status of all its incoming links are received, it evaluates its join condition (Links Semantics : page 106 of [18]).

- Event Handlers Enablement and Disablement Semantics : the event handlers associated with a scope are enabled when the parent scope starts (Enablement of Events : page 142 of [18]). An exception to this rule applies to scopes that contain a process initial start activity (for ex : receive). If a scope contains an initial start activity then the start activity MUST complete before the event handlers are installed (Scope Initialization : page 116 of [18]).

  When the primary activity of a scope is complete, all its contained event handlers are disabled. However, the running event handlers MUST be allowed to complete thus delaying the termination of the whole scope (Disablement of Events : page 143 of [18]). .

- Termination Semantics : the termination of BPEL activities may be forced to happen immediately (we call it strong termination) or may be given some time to be done (we call it weak termination). This depends on the type of the BPEL activity in question. For example, the `assign` activities are sufficiently short-lived that they MAY be allowed to complete rather than being interrupted when termination is forced while each `wait`, `receive`, `reply` and `invoke` activity MUST be interrupted and terminated prematurely (Termination Handlers : page 135,136 of [18]).

- Scope Hierarchical Termination : upon a fault signal, a hierarchical termination is adopted for BPEL scopes. The behavior of fault handling for scope C MUST begin by terminating all activities that are currently active and directly enclosed within C (Fault Handlers : page 132 of [18]).

- Hierarchical Fault Propagation : an unhandled fault in a scope is propagated to its direct enclosing scope's fault handler. Whenever a

<catchAll> fault handler (for any fault), <compensationHandler>, or <terminationHandler> is missing for any given <scope>, they MUST be implicitly created and implemented by a rethrow activity (Default Fault, Compensation, and Termination Handlers : page 132 of [18]).

2.3 Overview of the Transformation

The modeling in FIACRE is based on the structure of the BPEL process. The WSDL part defines the communication points of the BPEL process. These connections are made first by the definition of partnerlinks. A partnerlink is a communication exchange between two partners. The process is one of the partners, and another service is the other partner. A partnerlink defines the role that the process plays (if any) and the role that the partner service plays (if any) in the particular exchange. Depending on the role, a process can use the porttypes in order to send calls/results or receive calls/results.

Accordingly, the static part of BPEL (WSDL) is modeled in FIACRE as global types (see Section 2.1). As for the dynamic part consisting of the primary activity of the BPEL process and its associated handlers, it is modeled as an outermost FIACRE component containing the composition of the primary activity's corresponding FIACRE pattern with the FIACRE patterns of these handlers (Fig 2.1). Finally, the communication points of a BPEL process are modeled by two ports in FIACRE (`I` for input and `O` for output). These ports are typed with the global types modeling the WSDL part.

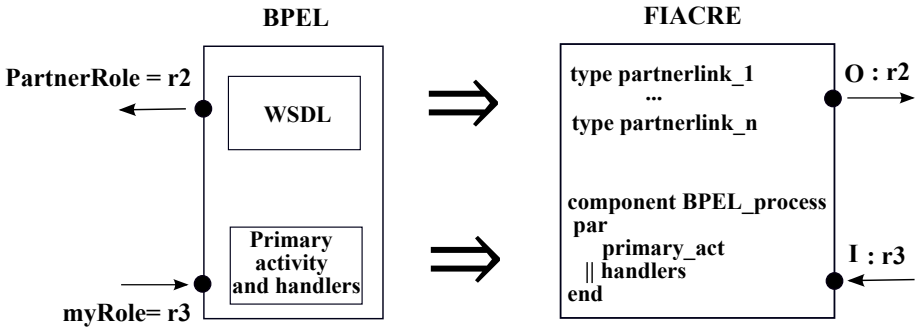


Figure 2.1 – *Transformation Structure*

This outermost component builds a parallel composition of component instances representing the BPEL nested activities. Moreover, BPEL basic activities are transformed to FIACRE processes while BPEL structured activities and handlers – being able to contain other activities – are transformed to FIACRE components.

2.4 Modeling the WSDL

The interaction with the environment is supported by partnerlinks. In BPEL, each partnerlink may contain two roles (`myRole` and `partnerRole`) typed with `Porttype`. Each of the `Porttype` declares

several operations used to receive (Input) or send (Output) messages (Fig 2.1).

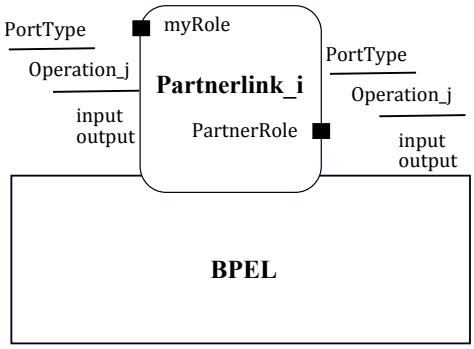


Figure 2.2 – *Connections of BPEL*

Consequently, this structure is modeled in FIACRE by creating two different enumerated types named `inputs` and `outputs` used to model respectively the inputs and the outputs of each operation. The type `inputs` (resp. `outputs`) will be the union of the type of :

- the `input` (resp. `output`) arguments of operations of the `myRole` of every Partnerlink (1).
- the `output` (resp. `input`) arguments of operations of the `partnerRole` of every PartnerLink (2).

More precisely, the types encoded in FIACRE in order to model the WSDL part are as follows :

1. Type `inputs` : The name of the constructor is built from the name of the partnerlink and a suffix `recv_call` (vs. `recv_result`) in case (1) (resp. in case (2)). As for the name of the constructor's type argument, it is built using the name of the partnerlink with the suffix `type_args` (vs. `type_results`) in case (1) (resp. in case (2)). The FIACRE code of the `inputs` type is shown in the following :

```
type inputs_BPELName is union
     partnerlinkName_recv_call of partnerlinkName_type_args
   | partnerlinkName_recv_result of partnerlinkName_type_results
   | ...
end
```

2. Type `outputs` : The name of the constructor is built from the name of the partnerlink and a suffix `send_call` (vs. `send_result`) in case (2) (resp. in case (1)). As for the name of the constructor's type argument, it is built using the name of the partnerlink with the suffix `type_args` (vs. `type_results`) in case (2) (resp. in case (1)). The FIACRE code of the `outputs` type is shown in the following :

```
type  outputs_BPELName is union
        partnerlinkName_send_call of partnerlinkName_type_args
      | partnerlinkName_send_result of partnerlinkName_type_results
      | ...
end
```

Similarly, we define the `mR_input_type` and `pR_output_type` family of types by considering the porttypes and eventually their operations.

Data Abstraction In our approach we have chosen to abstract from data since variables in BPEL may have an infinite data domain. Actually, with respect to model checking purposes, considering the actual values is not realistic because of state explosion which leads to an impossibility of verification. For this, a constant is created for each data type that represents the type of the message. So the operations will be typed by this constant that represent the BPEL type of the message. This abstraction leads us to a non-deterministic choice in some cases. Examples of such cases could be in BPEL activities that depends on data in order to make a decision (for example, *if,while* and *repeat Until*). On the other hand, if a treatment involves finite data domains of type boolean or enumerated types, these are preserved by the transformation to FIACRE. Particularly, in the case of the transition and join conditions related to links which have positive or negative status. These features will be detailed in further sections.

2.5 Behavioral Aspects in FIACRE

The transformation is guided by the constructs of BPEL. A BPEL process consists of multiple BPEL activities put together. Accordingly, each construct of the language is translated separately to a FIACRE component.

1. Basic activities will be translated to Fiacre `Process`.
2. Structured activities may embed other nested activities. Hence they will each be translated to a Fiacre `Component`.

In both cases whether an activity is translated into a `Process` or a `Component`, it will share a common interface. This interface will consist of the set of FIACRE ports and shared variables that each pattern should have in order to enable their composition. We will start by introducing a basic interface (Fig 2.3) containing 2 ports : S and F (start,finish) used to connect the activities in the composition. At the end of every activity, the finish port is synchronized with the start port of another. We will extend the interface progressively in the rest of the chapter.

2.5.1 Common Behavior of Activities in FIACRE

All of the activities in FIACRE share a common behavior. This behavior consists in modeling the outgoing and incoming links with their respective conditions (Fig 2.4: behavior with `suppJoinFailure = yes` or with `suppJoinFailure = no` [18]). Moreover, each activity will include additional control events and shared variables used in FIACRE for modeling

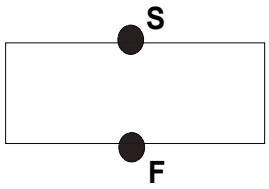
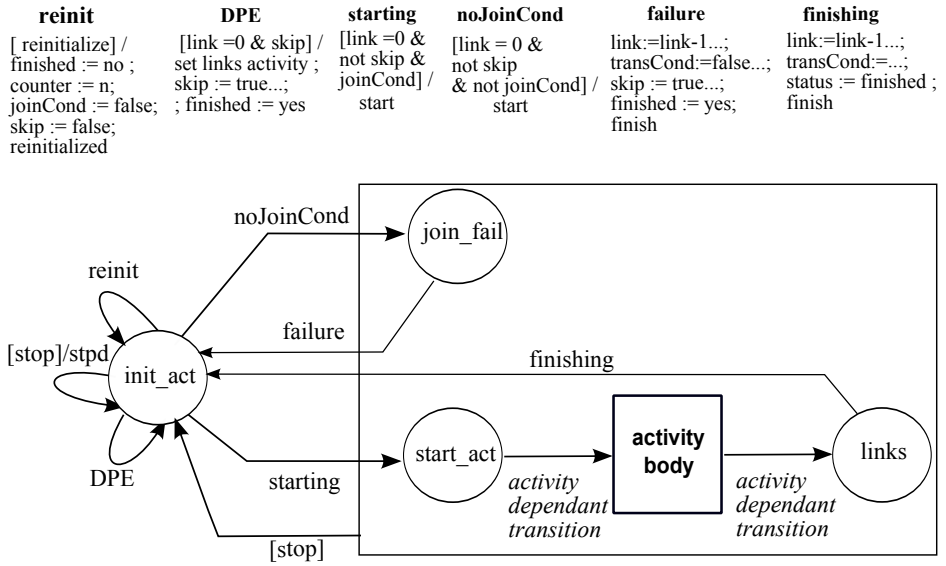


Figure 2.3 – *Common interface*

the forced termination of activities or for reinitializing the activity in case it is nested inside of a repeatable construct. In Fig 2.4, the activity body square represents a specific FIACRE pattern depending on the type of the modeled activity. This could be any basic or structured activity.



(a) Behavior with SuppJoinFailure = yes

(b) SuppJoinFailure = no : from join_fail joinFailureError ;to init_act

Strong Termination : reinit, DPE, starting, noJoinCond, failure , finishing, activity dependant transitions are prefixed with [! stop];

Figure 2.4 – *Common Behavior*

Composing the activities: Fig 2.4(a)(b) : every activity waits for its start signal by synchronizing on its `start` port. In the same way, at the end of the activity, it signals its end by synchronizing on the `finish` port used as the start of the subsequent activity.

Termination Modeling : Fig 2.4(a)(b) : we use a Boolean variable `stop` and a port `stpd` which signal respectively the termination demand and the occurrence of this termination. Once the `stop` variable evaluates to True, the activity may respond by transiting back to its initial state from which it would communicate its `stpd` port. Depending on whether a strong or a weak termination is adopted, an activity may stop immediately or not. On the contrary to some other techniques [60, 74], we consider here a strong termination. This is closer to the BPEL semantics (Termination Handlers : page 135,136 of [18]). Implementing a strong termination

means that whenever a stop is demanded, the activities are no longer permitted to execute. This can be done in FIACRE by adding explicitly the guard `on not stop` to each transition. Actually, a weak termination is only modeled for the assign activity (see assign).

Links Modeling : in order to model the BPEL links, the common interface is extended by four variables.

1. A *counter* variable initialized to the number of an activity's incoming links. At the end of every source activity it decrements the corresponding counter of the targeted activities by 1. As for the targeted activities, they will be able to execute whenever their counter evaluates to 0.

2. A Boolean variable modeling the transition condition: having disregarded data values in our analysis, we only consider three transition condition values. At the end of the source activity, an activity valuates its transition condition to either *true*, *false*, or *any*. The latter represents the transition condition that is either *true* or *false* and corresponds to the only case that the BPEL data could not be treated in FIACRE. In this case, the choice (between true or false) is nondeterministic.

   As for the incoming transition conditions, they will be tested in form of join conditions (Links Semantics Page 106 of [18]). These Boolean variables will be evaluated before the start of the targeted activities.

3. The *skip* variable : in the case of the normal execution of an activity, the value of this variable corresponds to *False*. However whenever this variable evaluates to *True*, this means the corresponding activity should be skipped following the rule of the Dead-Path-Elimination [1] because it belongs to a branch which is not executed anymore.

4. A Boolean variable *finished* that designate the status of the activity, whether it is finished or not. It is set to finished by the activity in the case that it has finished normally without any forced termination. It is also set to finished by the corresponding fault handler after a forced termination is realized. The knowledge of the status of the activity is important when dealing about setting the links of an activity. In fact in case of a fault, the links are not set by their own activities but by the enclosing scope that handles the fault. This is explained in the Fault Handler pattern.

A source activity signals its end by decrementing the counter associated to its targeted activities by 1 and by updating its respective transition conditions. The targeted activities are executed once their counter evaluates to 0 and their join conditions evaluates to *True*. Nevertheless, if the join condition evaluates to *False* :

[1] The dead path elimination is the result of two cases : (i) The join condition of an activity evaluates to False. (ii) If during the execution of a structured activity A, its specification says that an activity B nested within A will not be executed.

1. **suppJoinFailure = yes** : Fig 2.4(a) : the activity should set its outgoing links to False before terminating (`j-i`).
2. **suppJoinFailure = no** : Fig 2.4(b) : a fault *joinFailure* is thrown and no finish port is transmitted.

Furthermore, in case the activity in question is a structured activity, all its nested activities should also set their outgoing links to False but only after their incoming links are received based on the Dead Path Elimination rule. This is modeled by valuating the `skip` variable to *True* which is evaluated before the start of each activity.

Reinitialization Modeling : Fig 2.4(a)(b) : the same mechanim of termination is used to model the reinitialization (a shared variable `reinit` and a port `reinited`). When a reinitialization is asked, the activity resets all its control variables namely its links counter variables, transition conditions and its skip variable before synchronizing on its `reinited` port.

2.5.2 Basic Activities.

In order to model the basic activities, we increment the interface by the ports (`msg_in, msg_out`) used to communicate with the environment, the shared variable `vars` that represents the BPEL variables and the port `Fa` used to throw BPEL faults. Moreover, each of the basic activities - discussed later here- includes the common behavior. Therefore, these activities will implement the common behavior pattern already given all along with additional implementations related to the body of the activity.

Receive The *receive* activity does a blocking wait for a message to arrive. In the following pattern, the process associated to *receive* will wait for a message of type *call* on a known partnerlink, porttype and operation. Once the message is received, this message is either stored inside the corresponding variable of the *t_var_record* either a *F_variableError* is thrown. This fault designates the faults the BPEL process may throw (like invalidExpressionValue, invalidVariables, or uninitializedVariable). Because we have abstracted data values, the choice between the two possible transitions once the process is at the *copy_var* state is a non-deterministic choice. The Fiacre process pattern for *receive* is given in Fig 2.5 :

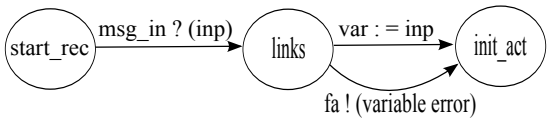


Figure 2.5 – *Receive pattern*

Reply Conversely to the *receive* activity, the *reply* activity will send a message of type *result* on a specific partnerlink, porttype and operation. It will at first extract the message to be sent from the corresponding variable in the *t_var_record*. If succeeded, the *reply* activity may continue otherwise the *F_variableError* fault is thrown.

The Fiacre process pattern for *reply* is given in Fig 2.6.

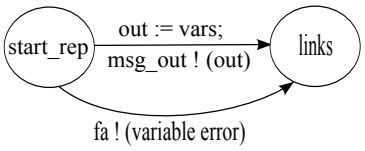


Figure 2.6 – *Reply pattern*

Invoke The *invoke* activity is used to call the partners web services. Both synchronous and asynchronous invocations exist. In synchronous invocations the BPEL process sends a call and waits for an answer of the called web service. An asynchronous invocation does not block the calling service and it continues after sending the call. Its pattern is exactly the same as the reply pattern. As for a synchronous invoke, it starts by extracting the message to be sent from the variable. It then awaits for a response to arrive. The response can be an incoming message or also a fault sent from the invoking partner. In the first case, the message is stored in the corresponding variable. In the case that a fault is received, the invoke will then throw an internal fault to be handled by the associated fault handler.

The Fiacre process pattern of the synchronous invoke in Fiacre is given in Fig 2.7.

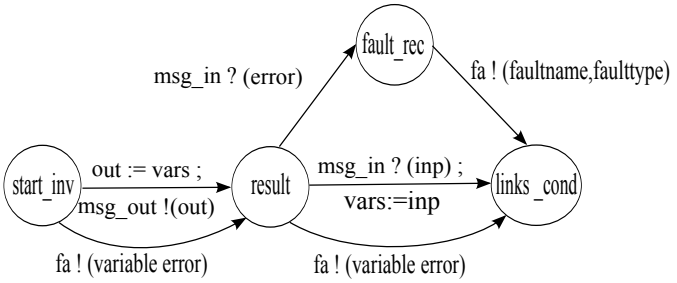


Figure 2.7 – *Synchronous invoke pattern*

Assign The Fiacre process pattern for *Assign* is given in Fig 2.8.

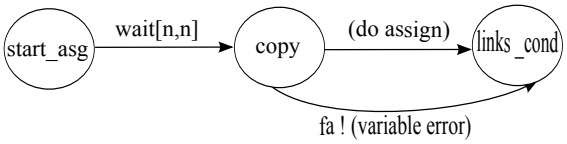


Figure 2.8 – *Assign pattern*

Since we do not model all data values, the *do_assign* in the code below will generate three cases :

1. The case that we assign Boolean variables in the form *Boolean_var1 := Boolean_var2 Op Boolean_var 3* with *Op* a Boolean operator. In this case, the assignment is copied as it in the Fiacre code.

2. The case that we assign a Boolean variable with a Boolean expression in the form *Boolean_var := Boolean_expression* and that we don't know

how to translate the Boolean expression in Fiacre. In this case, the Fiacre code will contain a non-deterministic assignment in the form of *Boolean_variable := any*.

3. The case that the variables in the assign activity are not Boolean, we assign the left hand variable with the default value associated to its type. This type information can be used by error handlers.

Moreover, in order to model a weak termination, we add an explicit wait n units of time before proceeding with the assign activity. This time corresponds intuitively to the data base access time. At the contrary of all other transitions, the transition labeled with this time interval is not prefixed with `on not stop` that forces an immediate termination.

Empty The *empty* activity is an activity that does nothing. It is modeled in FIACRE because it could be the source or the target of other activities. Its process corresponds Fig 2.4 with no body associated.

Throw In BPEL we can explicitly signal errors using the *throw* activity. The *throw* activity provides the name for the fault, and can optionally provide further information about the fault. A fault handler can use such data to handle the fault and to populate any fault messages that need to be sent to other services. In Fiacre, the type of a fault will be the name of the BPEL fault associated with a constant that represents the type of the BPEL variable in case it was present. The Fiacre process pattern of the *throw* activity is given in Fig 2.9.

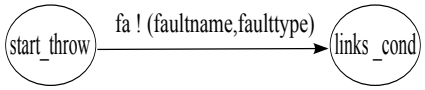


Figure 2.9 – *Throw pattern*

2.5.3 Structured Activities

Structured activities specify the order in which their nested activities are executed. The idea of the modeling in FIACRE consists in adding a Controller process that has no equivalence in BPEL. Based on the type of the BPEL structured activity, a different controller is created. This will specify the way the nested components of structured activities are executed (sequential, parallel, conditional...). Moreover, it will capture the incoming and outgoing links of the structured activity. The components of all of the structured activities in FIACRE consist of a composition of a controller process with their nested activities.

We use the following notations in the specification of the patterns of the structured activities : black dots designate ports, black square designate shared variables, a directed arrow between two ports means a transmission of the event, namely a synchronization, the ports and shared variables underlined means that this interface (of a upper component) will be

transmitted (for synchronization purposes) to the interface of an embedded component having the same color indicator, and the dashed squares inside a component means that the contents of this square are declared as local ports or variables to the component.

Sequence

The `sequence` activity executes its nested activities in a sequential manner. Fig 2.10 gives the sequence pattern of two activities.

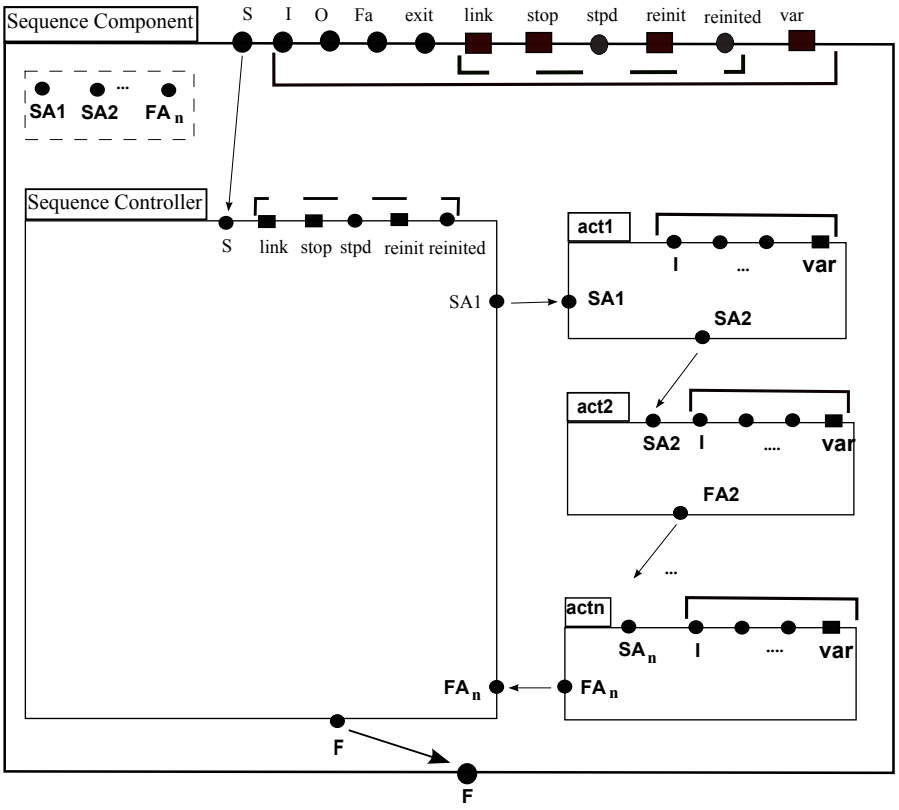


Figure 2.10 – *Sequence component*

Whenever the start signal of the sequence (port S) is received and all of the conditions hold, the sequence controller will signal the execution of the first activity in the sequence by transmitting its start signal (SA1). At this stage, the nested activities will run in a sequential order by synchronizing on their start and finish ports. The end of the first activity (FA1) as shown in the sequence component (Fig 2.10) will be the start of the second activity and so on. Meanwhile, the sequence controller will do a blocking wait until the last activity of the sequence signals its termination. Whenever the finish signal of the last activity is received (FA2 in Fig 2.10), the sequence controller will evaluate the outgoing links of the sequence and then transmits its finish signal (F) indicating the end of the sequence activity. The sequence controller of the sequence pattern is given in Figure 2.11.

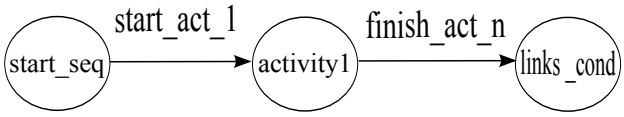
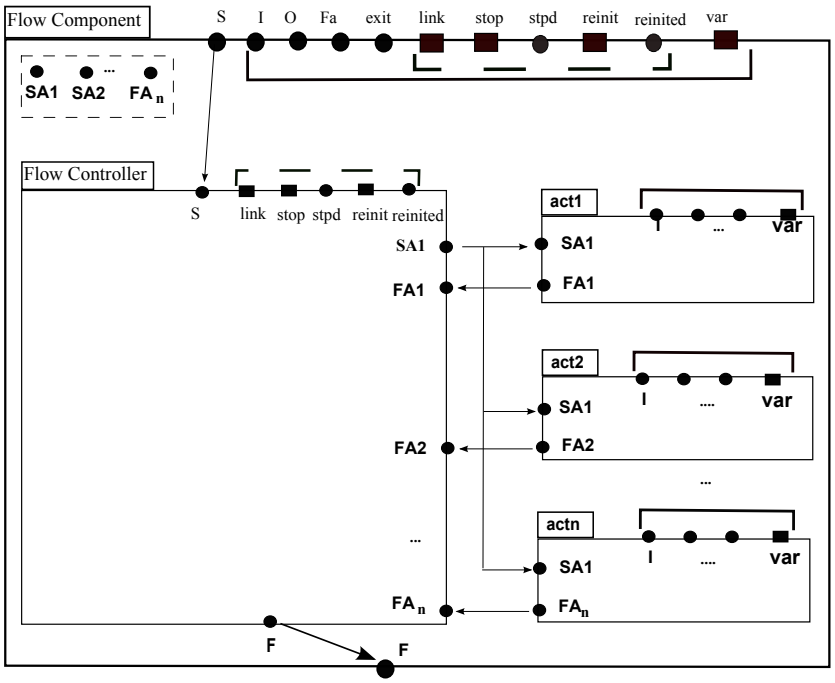


Figure 2.11 – *sequence controller*

Flow

The `flow` activity executes its nested activities concurrently. In Fig 2.12, the nested activities are composed with a flow controller that controls their concurrent execution. The controller (Fig 2.13) starts by signaling simultaneously the start of all the nested activities (`SA1`). Then, it does a blocking wait until the finish signals of all the activities are received asynchronously.

Figure 2.12 – *Flow component*



The flow controller makes use of a variable named `flow counter` that is initialized to the number of nested activities of the flow. Each time a `finish` signal of a nested activity is received, the flow controller decreases the `flow counter` variable by 1. Once all the finish signals are received (`flow counter` = 0), the flow controller will signal the finish of the flow.

Scope Component.

The scope component in FIACRE is given in Fig 2.12. Here we only model the primary activity of the scope, its event handler and its fault handler. In addition to this, a scope in BPEL is associated to a termination and a compensation handler. Here we abstract from the behaviors of these two. Fig 2.12 contains two sub-components that respectively define nominal execution and fault and stop handling. These components communicate

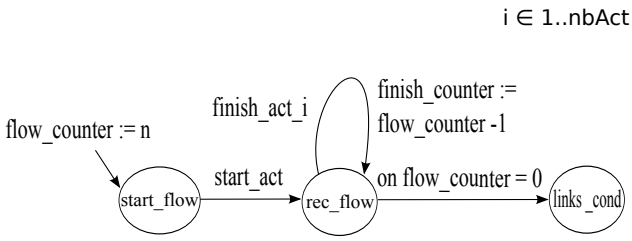


Figure 2.13 – *Flow controller*

with each other through local ports or variables, and they communicate with the scope environment through the scope component interface.

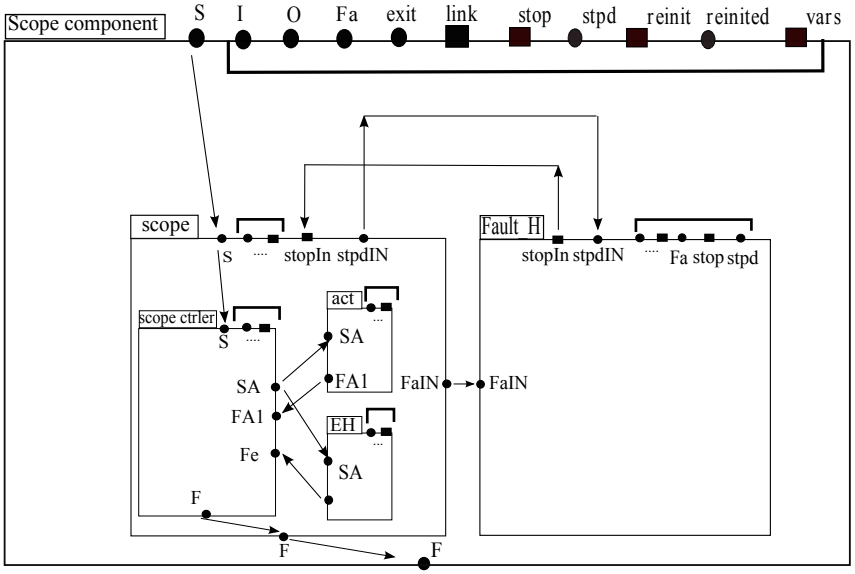


Figure 2.14 – *Scope Component*

Nominal Execution : It designates the execution of the first sub-component of the scope. The start signal (port S) of the scope component is intercepted by its scope controller `(scope ctrler)` leading to the execution of the primary activity of the scope and its associated event handler `(port SA)`. The end of both of these constructs leads to the end of the scope component `(F)`. The scope controller behaves the same way as a flow controller (Fig 2.13) where the variable `flow counter` is initialized to 2 corresponding to the scope's primary activity and its event handler.

Fault and stop handling : it designates the execution of the fault handler associated with the scope. The fault handler in FIACRE is executed as a result of two scenarios :

1. Fault thrown inside the scope : this is the result of a synchronization on the port `Fa_In`. If the fault can be handled by the fault handler, it will start by asking the termination of the scope by valuating the `StopIn` variable to True. Then, it will wait until all of the activities synchronize on `stpdIn`. After that, the fault handler will proceed by executing its activities before eventually synchronizing on the finish

port `F`. If the fault could not be handled, it will be propagated directly to the enclosing scope by synchronizing on the `Fa` port shared with the interface of the scope component.

2. Forced termination signaled by the enclosing scope : the `Stop` variable shared with the scope component is valuated to True. As a result, the fault handler terminates the activities of the scope with the same mechanism described earlier before signaling its termination to the enclosing scope by synchronizing on the `stpd` port shared with the scope component interface.

2.5.4 BPEL Handlers

In this chapter, we only treat the fault and the event handlers since they will appear later in the use case. For the interested readers, the termination handler and the compensation handler are given in Appendix B.

Fault handlers

The treatment of faults in BPEL are done by fault handlers. Each scope as well as the BPEL process have its own fault handler associated. Fault handlers may be of two types:

- A default fault handler which will be created in case no user defined fault handler was specified. A fault caught by this fault handler triggers the compensation of the direct child scope before being propagated to the next enclosing scope. But since the compensation is not modeled in this chapter, we say that the fault is directly rethrown (Full Fault Handler in Appendix B).
- A user defined fault handler in which multiple *catch* branches and an optional *catchAll* branch are specified. Each of the *catch* branches as well as the *catchAll* are associated to a nested activity that is executed when one of the branches is chosen. The nested activity will execute whenever a match is found. In case no match is found :
  - *catchAll* is specified : the *catchAll* branch will be chosen and then its associated activity is executed.
  - *catchAll* not specified : The fault handler will behave the same way as the default fault handler.

In the transformation to Fiacre we will differentiate between two types of Fault Handlers. One with a *catchAll* branch specified and the other without it. A default fault handler is a variant of the *No catchAll* version where *No catch* branches are specified. The fault handler component is depicted in Fig 2.15. A fault handler starts whenever a synchronization on the port *FaIn* takes place. Depending on the received fault type, a *catch* branch is chosen and the activity of the branch is executed (SA1 or SA2 and FA1 or FA2). We note particularly, that in case of an unhandled fault, or a fault thrown inside the nested activities of the fault handler, the fault will be propagated to the next enclosing scope Fault handler by synchronizing on the *Fa* port.

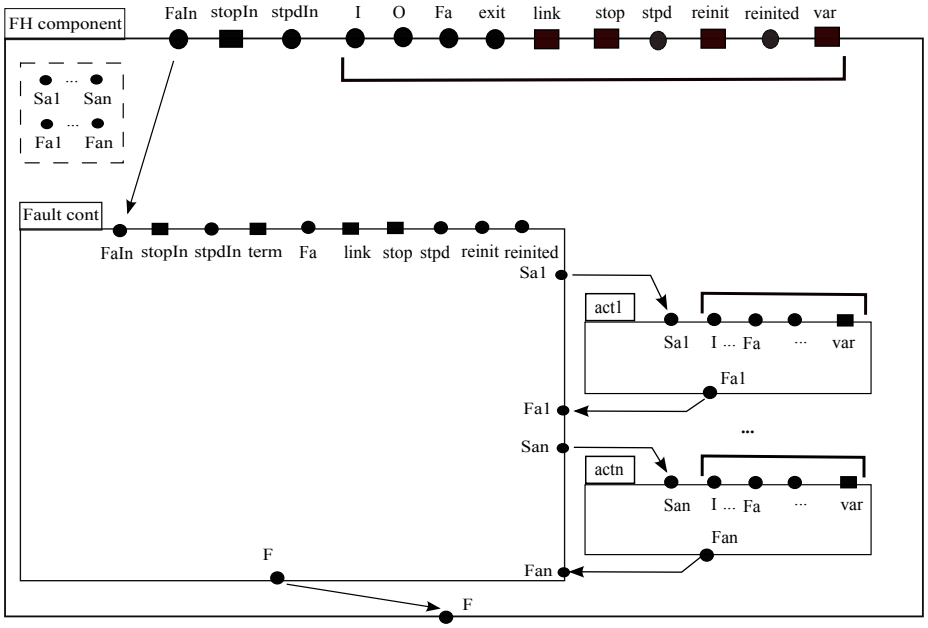


Figure 2.15 – *Fault handler component*

Two versions of fault handler controllers are given depending on whether a *catchAll* branch is specified or not.

Fault handler controller : no catchAll Fig 2.16 : at the *start_catch* state, the controller will wait for either a fault signal on the port *Fau1* or a termination demand from its upper scope (the scope that encloses the scope to which the fault handler is associated). In case of a fault signal, then depending on whether the fault is matched with one of the *catch* or no, the controller will transit to one of the two states *catch_bdy* or *not_handled*. Here we remind the reader that the fault matching in Fiacre will be based on the name of the BPEL fault and on the constant that is associated with the fault name which identifies the type of the variable associated with the BPEL fault.

1. At the *catch_bdy* state the controller signals that a fault has been detected by updating the error variable with the value `other` and the `stop` variable to true. After receiving the `stpd` event, the fault handler controller then executes the nested activity associated with the matched fault by signaling its start event and by updating the value of a local variable `branch` with an integer that points to the chosen branch before transiting to the `links_cond` state. This `branch` variable will be used to indicate which branch has been chosen by the controller, so that the other activities that are associated to the other branches are informed that they are no longer activated. In this case, positive values of the `skip` variable are transmitted to the unchosen activities. At this point, the fault handler will proceed by evaluating the status of all the activities nested in the scope (not just the direct nested activities). In the case the status was different than *finished* -meaning that the outgoing links of the activity were not evaluated- the fault handler controller will decrement their counters and will

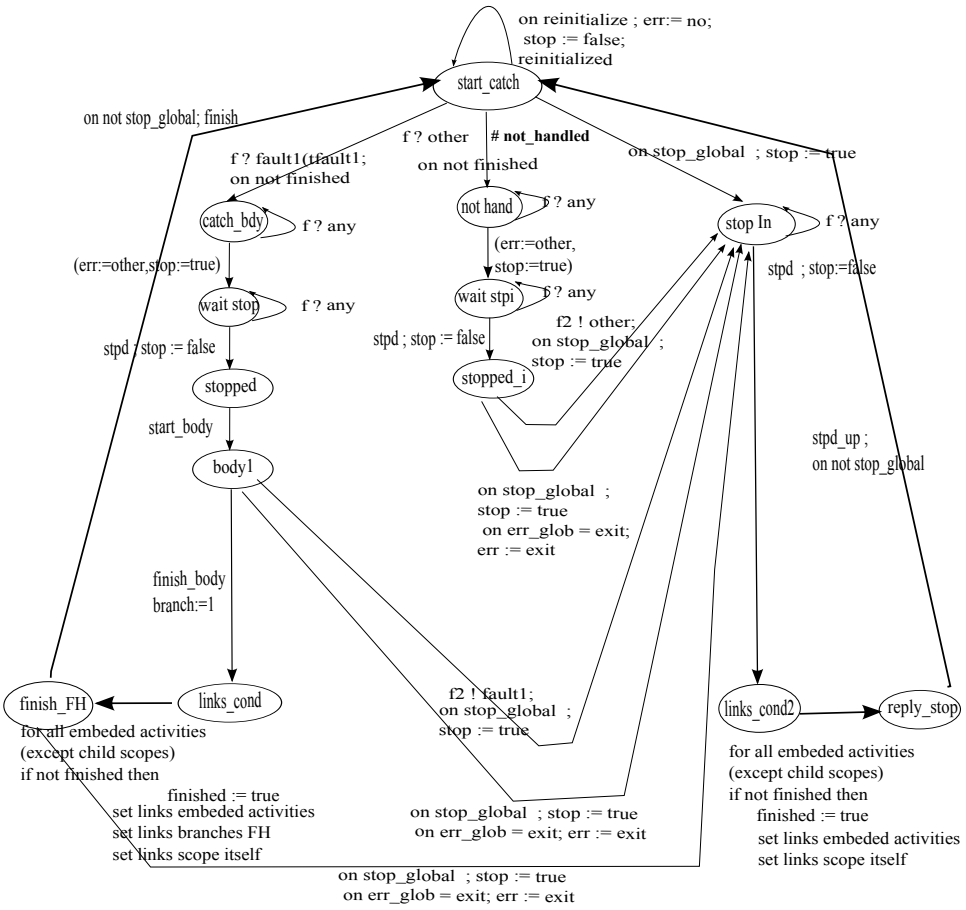


Figure 2.16 – *Fault Handler Controller : No CatchAll*

set their conditions to False. Furthermore it will set their status to *finished* to indicate that their outgoing links have been evaluated. As for the links of the scope (the scope to which the fault handler is attached), its outgoing links will be evaluated normally.

2. At the *not_handled* state, the controller will have the exact behavior concerning the termination of activities. It will then proceed by propagating the fault to the next enclosing scope using the *Fau2* port.

Finally, in case of a termination demand made by valuating the `stop_global` variable to True, the fault handler modifies its stop variable to true and transits to the `stop In` state. At this state, the fault handler proceeds by stopping the activities of its associated scope and then by signaling a stopped signal (`stpd_up` event) to the upper scope.

Fault handler controller : catchAll The same explanation goes for this controller. The only difference is that a fault is not rethrown to the next enclosing scope, since a *catchAll* has been specified.

Event Handler

Each scope, including the process, can be associated to a set of event handlers. These event handlers are run concurrently and are invoked when

the corresponding event occurs. There are two types of events. First, events can be inbound messages. Second, events can be alarms, that go off after user-set times [18]. Event handlers are executed as long as their attached scope is in normal execution. When the primary activity of the scope finishes faultlessly, the event handler should also be finished. Whereas different incoming messages or timeout events are handled concurrently, if two messages for a single event handler branch arrive concurrently, they are handled sequentially. We give the component of the event handler in Fig 2.17.

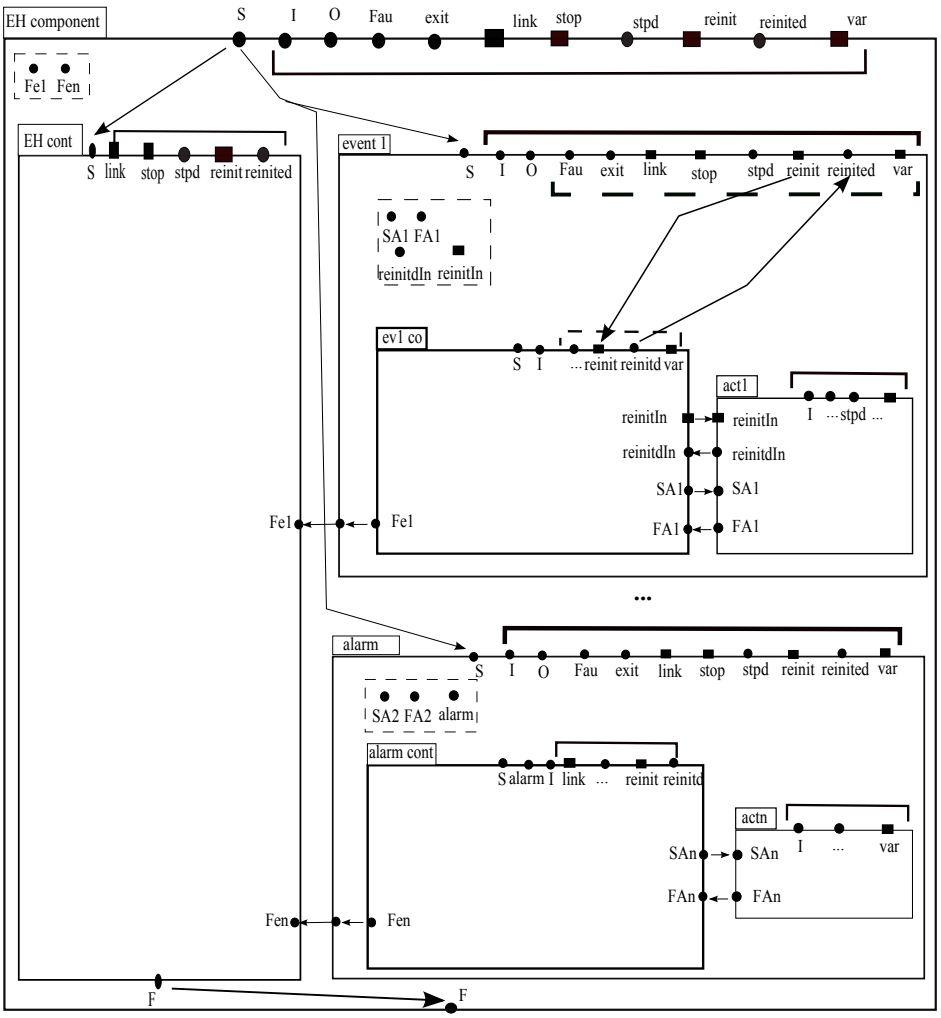


Figure 2.17 – *Event handler component*

Fig 2.17 : once the start signal is received by the event handler component (S coming from the attached scope), the start signal is transmitted concurrently to the eventHandler controller at one hand and to all of the other entities (event1,...,alarm) that represent the branches of the event handler. Here we assume that the scope to which the event handler is associated does not contain a process initial start activity. [2].

[2]Otherwise, it is wrong to start the event handler with the start of its scope. It this case it needs to be started after the finish of the start activity (receive). This is modeled in our patterns by a counter variable that is decremented upon the finish of the receive (see Chapter : Use Case)

Once the event handler controller (Fig 2.18) receives its start signal, it will enter a waiting state in which it will wait for the reception of the finish signals (Fe1,Fen) of all the branches of the event handlers. At the time these events are received, the event handler controller is ready to transmit the finish signal (F) of the event handler component indicating the end of the event handler.

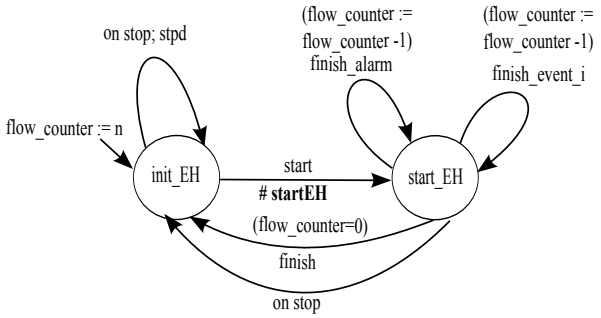


Figure 2.18 – *Event handler controller*

As for the branches of the event handler (event1,alarm), once they receive the start signal (S) they will transmit it to their controllers in order to wait for a message or for a timeout to occur [3].

Message Event controller
Fig 2.19 : at the state `init_event`, the controller waits for the start signal to be received in order to transit to `start_event` state in which two transitions are possible :

1. The controller checks the scope's `finished` status. If it evaluates to True, the controller directly transits to the `finish_event` in order to transmit its finish signal back to the event handler controller (Fig 2.18).

2. Alternatively, the controller does a blocking wait until an input messages is received. In this case it will transit eventually to the *body1* in which the same test of the conditions of termination of the attached scope is again evaluated. In the case the scope is still in normal execution, the controller will transmit the start signal (`start_body1`) of the attached activity and then transits to the *body1_started* state. At this point the activity can no longer be stopped and is allowed to finish even if the scope has stopped.

   Once the body is finished (`body1_started` state), the controller either finishes (`finish_event` state) if the whole scope has already finished, or the controller reinitializes its nested activities and waits for eventually another message to arrive (`start_event` state).

Finally we say that at each state, the event handler can be stopped whenever a stop demand has been asked.

[3]Here we also note that the timeout treated is the relative time of BPEL. For the treatment of the absolute time, refer to Section 2.5.5.

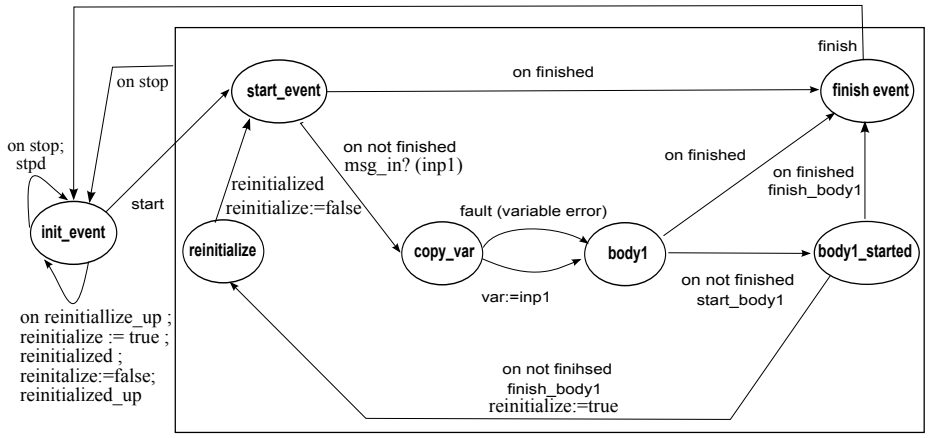


Figure 2.19 – *Event branch controller*

Alarm event controller
The events of the alarm branch in BPEL can be of 2 types :

1. The `for` and `until` expressions. An alarm branch specified with the `For` or the `Until` expression permit a single execution of the nested activity.

2. The optional `repeatEvery` expression in which we also specify a duration. With the `repeatEvery` expression the alarm will be fired repeatedly each time the duration period expires while the parent scope is active [18]. If the *repeatEvery* expression is specified alone, the clock for the very first duration starts at the point in time when the parent scope starts. If the `repeatEvery` expression is specified with either the `for` or the `until` expression, the first alarm is not fired until the time specified in the `for` or `until` expression expires; thereafter it is fired repeatedly at the interval specified by the `repeatEvery` expression [18].

In the following we will give the alarm of the For attribute and the alarm of the repeatEvery attribute. The case of the Until attribute is discussed in Section 2.5.5.

1. In the case of the For controller (Fig 2.20). At the point that the controller is at the `start_alarm` state, it will wait for a timed event to arrive. Since this is a relative time, the timed event is associated with the time interval `[`duration,duration] where `duration` is the value taken from the BPEL code.

2. Since the notion of time in the `RepeatEvery` is relative, it will be treated the same way as in the case of the `For duration`. So a timed event will be created and associated with the interval `[`duration,duration]. Furthermore, the controller of the `RepeatEvery` (Fig 2.21) will allow the repetitive execution of the nested activity.

Now we model the case that the *repeatEvery* expression is specified with the *for* expression (`Until` expression see Section 2.5.5). For this purpose, slight modifications to both of the alarm controller given in Fig 2.20

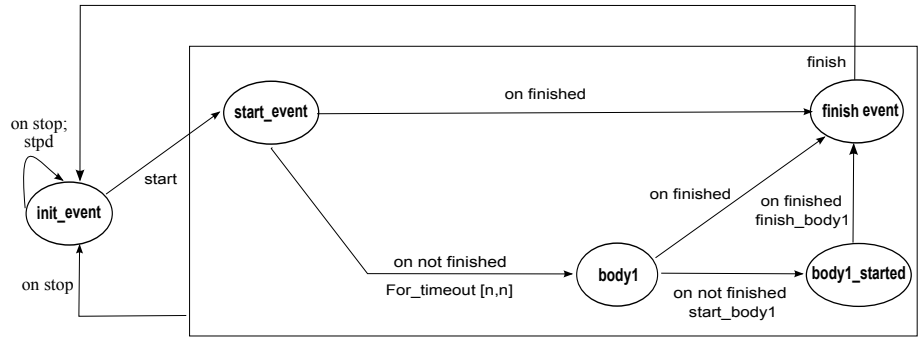


Figure 2.20 – *For Alarm controller*

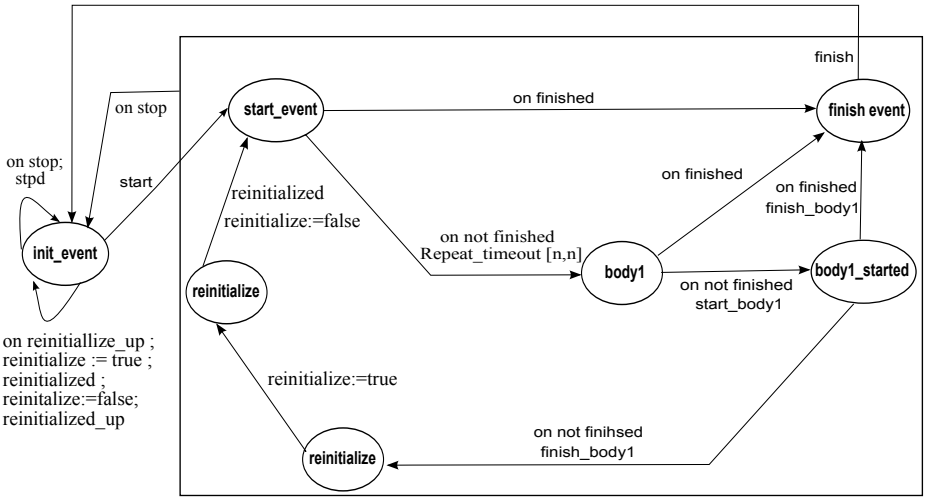


Figure 2.21 – *RepeatEvery Alarm controller*

and 2.21 are made. These modifications are shown in Fig 2.22. In the controller of the For (Fig 2.22 (a)), we add to the process an intermediary state between the *start_alarm* and the *body1* states, in which the controller will evaluate a Boolean variable *For_done* to True once its alarm has expired. This Boolean variable will be then tested by the controller of the *RepeatEvery* before it could start its alarm as shown in Fig 2.22 (b). We just note finally that this Boolean variable should be declared locally -because it will be shared by the controllers of *repeatEvery* and *For\Until*- in the event handler component given in Fig 2.17. Furthermore, it is reinitialized to False by the the *repeatEvery* controller once the event handler is asked to be stopped (Transition from *body1_started* to *finish_alarm*).

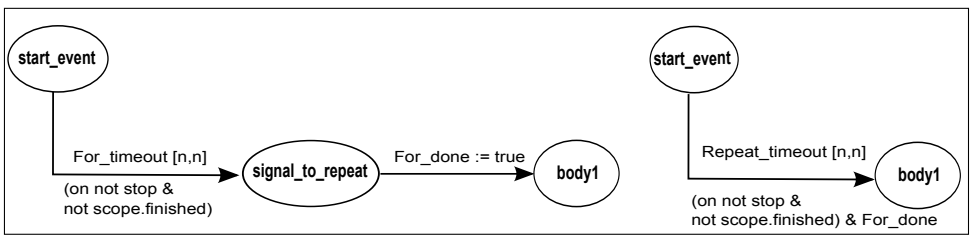


Figure 2.22 – *RepeatEvery Alarm controller specified with For/Until expression*

2.5.5 BPEL Timed Aspects in FIACRE

As we have seen in several constructs before, the modeling of the relative time of BPEL (`For duration`) is straightforward in FIACRE and it could be viewed as a timed event with the same minimal and maximal bounds. Concerning the absolute time of BPEL (namely the Until deadline), it could be handled by explicitly fixing the starting time of the process. In this way, we can easily bound the interval associated with the timed event to : [deadline - start , deadline - start]. Still, this is hardly a practical solution -especially in the context of verification- since one could argue that the starting date is not always known. In fact, we are interested in verifying the correctness of the system for any starting date.

Modeling of the Absolute Time : The idea is to add to the model a transition which makes it possible for a non-deterministic period of time to elapse between a reference date and the start of the system. This is done [69, 93] easily in formalisms based on timed automata. A clock `H` measuring the absolute time is introduced. An initial transition reinitializes all the clocks of the system except `H`. Moreover, this clock is tested in guards ($H \geq D$) in order to know if the absolute time D has been reached. In FIACRE, we do not have explicit clocks, which makes this situation harder to handle. In order to model the absolute time in FIACRE, the two following problems should be considered :

1. Model a non deterministic elapse of time initially: a synchronization on a timed port is introduced.
2. Test whether the absolute time has been reached. This is done by introducing a synchronization on the port `after` being managed by a timer process. Once the deadline is reached, this synchronization becomes non-blocking

The system is built by composing the timer with the component modeling the root activity of the BPEL process:

```
component BPEL_Process is
    port after : none
    par
          after → process_act [after,...,]
       || after → timer [after]
end
```

Managing a Unique Absolute Time The `Timer` process runs concurrently with the real system and measures when the absolute deadline is reached. We will start by treating the case of a system having one absolute time. The behavior of the process is as given in Fig 2.23 :

In order to model the non deterministic delay, we identify two cases in the timer process depending on whether the absolute date has been reached or not. After choosing one of these cases, the process will either wait until the time is reached in the former case or enable an immediate synchronization on the `after` port in the latter case.

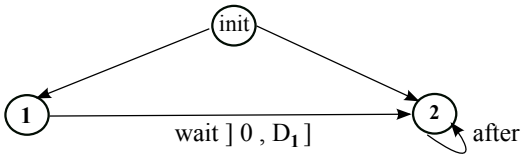


Figure 2.23 – *Absolute time treatment*

Managing Multiple Absolute Time Now that the idea is clear, we will generalize the timer process (Fig 2.24) so that we can take into account the case when several absolute times are used in the BPEL description.

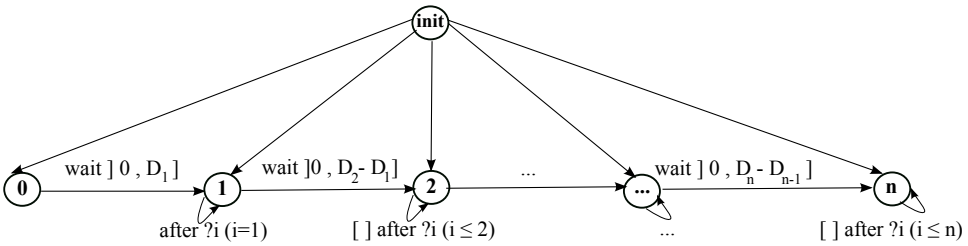


Figure 2.24 – *Absolute time treatment*

In Fig 2.24, all the absolute times of the system are sorted chronologically. Initially and non deterministically the process is set up to one of the time intervals (i.e strictly before the first, exactly or after the first...). Afterward, based on what interval has been reached, the process enables a synchronization on the $\texttt{after}_i$ ports associated to all the elapsed dates.

Absolute Time in BPEL Constructs In all the BPEL constructs we have seen until now, we have only treated the relative time. Namely in the wait activity and the alarm event of the event handler. Now with the handling of the absolute time, the patterns of these activities need to be adapted. This simply consists at replacing the former timed event with the `after` event (Fig 2.25). These patterns will communicate with the `after` event of the `timer`. The communication is synchronous between the timer process and the root component in which all the FIACRE patterns are embedded. However the communication on the `after` event is asynchronous between the patterns themselves.

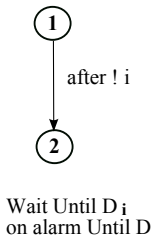


Figure 2.25 – *Absolute Time Adaptation*

2.6 Correctness of FIACRE Patterns

In this section we give our correctness criteria for the BPEL semantics properties given in Section : 2.2. The correctness is based on describing the semantics properties in the form of LTL properties. These properties can be either verified on isolated patterns or on instances of the FIACRE patterns. Here we follow the second technique and verify the correctness properties directly on instances of the use case's patterns (see Chapter Use Case). The satisfaction of the properties is a correctness proof of the preservation of the BPEL semantics by the FIACRE patterns.

2.6.1 Control Correctness

In the first category of properties, we consider the BPEL constructs while ignoring the links and the possibility of faults. This means that the order of execution of an activity depends only on the control flow and not on its incoming links.

Sequence The correctness of the start of the sequence activity may be given as follows :

$$\neg start_{act_1} \mathbf{W} start_{sequence} \qquad Sequence\ Start$$

That is, the first activity of the sequence cannot start unless the whole sequence is started. Here we use the weak until operator **W** in order to say that $start_{sequence}$ may not occur. We could have also generalized this property as $\neg start_{act_i} \mathbf{W} finish_{act_{i-1}}$. But this property is trivially verified in our patterns since $start_{act_i}$ and $finish_{act_{i-1}}$ are the same event.

A sequence ends when its last activity does. For n designating the last activity of the sequence, the end of the sequnce can be written as :

$$\neg finish_{sequence} \mathbf{W} finish_{act_n} \qquad Sequence\ Finish\ 1$$

That is, the whole sequence cannot finish unless the last activity of the sequence is finished.

Another type of formulas can be tested also, which is the reverse of the previous formula. The end of the last activity of the sequence eventually leads to the end of the sequence in case no forced termination occurs.

$$\Box(finish_{act_n} \Rightarrow \Diamond(finish_{sequence} \vee stpd)) \qquad Sequence\ Finish\ 2$$

Flow A flow starts when all its embedded activities do. Formally, for $i \in 1..NbActFlow$ the correctness of the start of the flow can be written in the following way :

$$\neg start_{act_i} \mathbf{W} start_{flow} \qquad Flow\ Start$$

That is, the activities of the flow cannot start before the flow does.

The end of the flow may be written as follows :

$$\bigwedge_i \neg finish_{flow} \mathbf{W} finish_{act_i} \qquad Flow\ Finish$$

That is the flow cannot finish before all of its nested activities do.

2.6.2 Links Semantics

In this category, we show how we analyze the semantics of the BPEL links.

An activity starts if the status of all its links have been received. Since the links are represented by a decrementing counter in FIACRE, this is checked by the following property :

$$\Box(start \Rightarrow act\_counter = 0) \qquad Incoming\ Links$$

This means that the start of an activity can only be made if its incoming links counter equals 0.

Concerning the setting of the outgoing links, the property that an activity cannot finish without setting its outgoing links (in case they are present) is trivially satisfied by our patterns since this is done in the same transition. Blocking (resp. firing) this transition means blocking (resp. firing) the finish signal and the decrementing of the counter corresponding to the outgoing links. Here remind the reader that in case of fault thrown while setting the outgoing links, a finish event is not signaled.

Sequence With Links Now considering the links, the Sequence and the Flow properties are adapted accordingly. Actually in the following properties we take into account both the links and the faults. For example, the properties of the Sequence become the following :

$$\Box(start_{sequence} \Rightarrow \Diamond(counter\_act_1 = 0 \Rightarrow \Diamond(start_{act_1} \vee fa))) \quad Sequence\ Links$$

That is, if the first activity of the sequence has received all its links and no faults (`fa`) are generated then the start of sequence implies the start of the first activity of the sequence.

2.6.3 Weak Termination

Being a weak termination, the stop demand is not considered instantly. This means that the activities have the choice whether to take the stop demand into consideration by transiting back to the `init` state or to remain in their active states. This can be represented by :

$$\Box(stop_{scope} \Rightarrow \Diamond stpd_{scope}) \qquad Weak\ Termination$$

That is, if a stop demand is signaled, the scope will *eventually* do its `stpd` event. We note here that this characterization of Weak Termination of BPEL is different than the Weak Until Semantics in LTL, since weak termination usually means that it may never happen.

2.6.4 Strong Termination

In the strong termination we want to guarantee that whenever a termination is demanded it will be taken immediately into consideration. Formally the strong termination can be written as follows :

$$\Box(stop_{scope} \Rightarrow (\neg \bigvee_{e \in event_{scope}} e) \mathbf{U}\ stpd) \qquad Strong\ Termination$$

That is, whenever a stop is demanded, none of the events of the scope may occur before the occurrence of the `stpd` event.

2.6.5 Hierarchical Termination

Whenever a scope is asked to stop, the fault handler of the scope starts by synchronizing with all the child activities nested in the scope. Once this synchronization has occurred -meaning that these activities have stopped- the fault handler controller proceeds with signaling the termination of the whole scope by synchronizing with its upper scope's fault handler. Hence a hierarchical view of termination is respected in our implementation. More formally, for i denoting the activities nested in the scope, the termination can be written in the following way :

$$\Box(stop\_up \Rightarrow \Diamond(stpd\_up \wedge \bigwedge_i init_i)) \qquad \textit{Hierarchical Termination}$$

That is, whenever a global stop is demanded, the scope (its fault handler) signals the `stpd_up` event and all the nested activities are in their initial states (where the local `stpd` event is available).

2.6.6 Hierarchical Fault Propagation

The fault propagation can be written in the following way :

$$\Box(\#not\_handled \Rightarrow \Diamond fault\_up) \qquad \textit{Fault Propagation}$$

This means that in case of receiving an internal fault via the firing of the `not_handled` transition of the fault handler, then the `fault_up` event will eventually be signaled. The keyword # is used in FIACRE to give an alias to the transition. So, *not_handled* is the name of a fault handler transition that leads to the `not_handled` state after receiving a fault that cannot be handled by the fault handler.

2.6.7 Event Handlers

The enablement and the disablement of an event handler depends on the nominal execution of the scope. In the following we give their correctness criteria :

Enablement : If the scope does not contain an initial start activity, the event handler is enabled upon the starting of the scope.

$$\Box(\#startEH \Leftrightarrow \#startActScope) \qquad \textit{Enablement Event}_1$$

That is, starting the scope means starting the event handler as well. *startEH* and *startActScope* are the aliases of respectively an event handler transition that starts the event handler and a scope transition that starts the primary activity of the scope.

Otherwise, the event handler is enabled once the inital start activity (receive) is finished :

$$\Box(\#startEH \Leftrightarrow \#finishReceive) \qquad \textit{Enablement Event}_2$$

Disablement : We give two disablement properties :

$$(\neg finish_{scope}\ \mathbf{W}\ finish_{Pact}) \wedge (\neg finish_{scope}\ \mathbf{W}\ finish_{EH}) \qquad Disablement\ Event_1$$

That is, the scope cannot finish unless both of its primary activity and its event handler are finished.

$$\Box(((finish_{Pact} \wedge \Diamond finish_{EH}) \vee (finish_{EH} \wedge \Diamond finish_{Pact})) \Rightarrow \Diamond(finish_{scope} \vee fa))$$

That is, if the scope's primary activity and the event handler finish in whatever order and at different instants then the whole scope eventually either finishes or a fault is thrown.

2.7 Conclusion

In this chapter, we have given the idea of the transformation from BPEL to FIACRE. This transformation is a one-to-one transformation, meaning that for each BPEL construct corresponds a FIACRE component. The transformation preserves the hierarchical composition of BPEL and treats its timed features. Another important feature of the transformation is that it can be proven correct via a verification of some BPEL semantic properties on the FIACRE patterns. In the next chapter, we present our BPEL verification framework.

Chapter Three

Verification Framework

3.1 Introduction

In this chapter, we discuss about the supported properties in our verification framework. This discussion mainly consists either in describing the properties already integrated and supported by our model checker or in describing techniques that are used in order to verify properties that are not supported.

3.2 Behavioral Verification with Tina

The verification is done using the Tina tool. Tina is particularly well suited to the verification of systems subject to real time constraints. The properties to be verified are often described in temporal logic, such as linear temporal logic (LTL). The result of the verification may lead to an accepting status, meaning that the model of the system satisfies the requirements, or exhibit an error. In the last case, it is often possible to extract a counter example, which is an explanation at the level of the model (generally an execution trace), which leads to a problematic state. In the Tina toolbox some of the verified properties are natively supported. However others need an additional treatment for their verification. This is going to be subject of the coming sections.

3.3 Supported Properties

The transformation patterns are used for verification purposes. Our verification framework supports four kinds of properties.

1. Temporal properties : these are the typical properties written in SE-LTL.
2. Data-Related Properties : in case the BPEL data types are finite, they are preserved in the BPEL/FIACRE transformation. It is possible then to verify properties that involve these data.
3. Structural properties : they express the "well-formedness" of the BPEL code. For instance, we are able to verify that a receive activity

is always matched with a reply activity, or that two concurrent receive activities cannot wait for the same operation. This is done by verifying statically that the faults resulting from such situations are not thrown.

4. Timed properties : they could be written in MITL [16] which is a timed variant of LTL and are handled here using timed observers and a temporal property, thus encoding such properties as reachability properties.

3.3.1 Temporal and Data-Related Properties

These are the properties that are supported natively by TINA. Examples of temporal properties are the boundedness, deadlock freeness, liveness and safety properties. These are verified directly by the model checker `selt` of TINA and are written in the SE-LTL logic.

Data-related are also supported by TINA. More precisely properties that manipulate or checks the value of a variable (i.e If condition for example) may be verified by querying or testing the value of these variables.

3.3.2 Structural Properties

Structural properties describe BPEL-specific requirements. Such requirements are independent of the user requirements and are verified by the BPEL engine dynamically. An example of such requirements is the order of execution of the BPEL receive and reply activities. In fact in BPEL, whenever a reply activity is not matched with a receive activity, a fault is thrown dynamically. Another case of such thrown faults is whenever multiple concurrent receive activities are used to wait for the same message. In this case, a fault is thrown because a conflict emerging from the choice of the receive activity to be executed would then take place.

In our verification framework, we are able to verify such properties via an instrumentation of the transformation itself. The `receive` and the `reply` patterns in FIACRE are then modified in order to allow such verifications. In order to model these aspects, the idea is to save the communication endpoint (`partnerlink`,`porttype`,`operation`) whenever a receive activity is executed and then to compare this saved information with the information of successive `reply` or `receive` activities. For this purpose, for each tuple (`partnerlink`,`porttype`,`operation`) a status variable of enumerated type is created. The enumerated type contains the following constants:

1. `wait_receive` : describes the fact that a `receive` activity is yet to be received.
2. `wait_reply` : describes the fact that `reply` activity is yet to be received.

Adaptation of the receive and the reply

The `receive` and `reply` activities are thus adapted w.r.t the newly created enumerated type.

The behavior of the status variable is as follows (Fig 3.2):

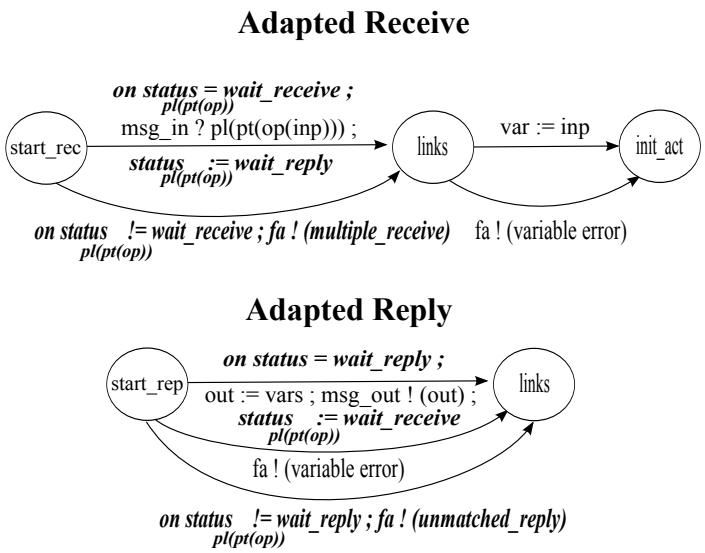


Figure 3.1 – *Adaptation of the receive and the reply patterns*

1. The status variable is initialized by `wait_receive`.
2. At the start of a `receive` activity, the status is tested. If it evaluates to `wait_receive`, it is updated to `wait_reply`. Otherwise a `multiple receive` fault is thrown meaning that a concurrent receive that awaits on the same connection is executed.
3. At the start of a `reply` activity, in case the status is different than `wait_reply`, the fault `reply_unmatched` is thrown meaning that a `reply` activity is found without having executed a `receive` activity previously.

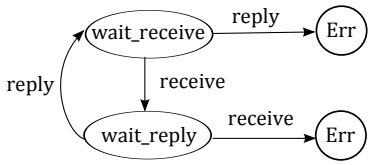


Figure 3.2 – *Status Behavior*

After the modification of the `receive` and the `reply` patterns, we are now able to observe the `multiple receive` and `reply_unmatched` faults. This observation makes it possible to verify the occurrence of these faults statically. This is done by using an event-based LTL formula that verifies that such faults are always avoided. This could be written in LTL as $\Box\neg multiple\_receive$ and $\Box\neg reply\_unmatched$.

3.3.3 Timed Properties

Time Embedding in BPEL Constructs

In order to enhance the timed aspects of a BPEL composition and to handle some QoS properties, it is interesting to assume some additional timing aspects on the BPEL constructs. In our verification framework we make some

default assumptions. First we assume that the elapsed time that represents the waiting for the reception of requests is unbounded. Accordingly, we associate a time interval of $[0, \infty[$ with the synchronization events that model a reception from the environment namely the `receive` activity and the `<on message/>` events of both the pick activity and the event handlers (`port: Input`).

Moreover, the BPEL code may be annotated with timing values in the form of the attribute `td = interval of time` (as done in the ITEmIS project [2]). Such annotations are used as an attribute of the synchronous invoke in order to give insights of the expected delay associated with the reception of responses resulting of previous invocations.

Moreover, we say that each activity may be annotated explicitly with its duration in the form of `td = interval of time`.

The modeling in FIACRE of such time constraints consists in adding an explicit `wait` instruction after the start of each activity. In case of synchronous invoke activity the `wait` instruction is added after the port associated with the output call of the invoke. Apart from these constructs, all of the other events are considered as instantaneous (Fig. 3.3).

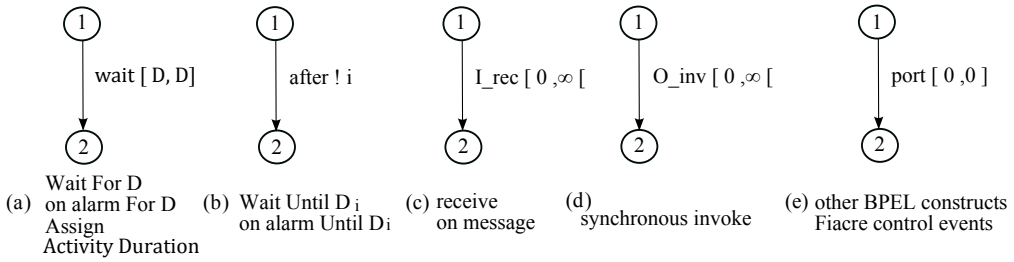


Figure 3.3 – *Summary of timed constructs in FIACRE*

The BPEL code may be annotated with other more global constraints (ITEmIS project [2]). Two main global constraints are used to annotate the BPEL code. A Process Duration constraint which specifies that the duration of the whole process is bounded by a certain time or a Duration between two activities constraint which specifies that the delay between two activities is bounded by a certain time.

Timed Observers

The verification of timed properties is done by timed observers. There exists two ways of modeling these observers :

1. Incorporate the timed observers at the BPEL level. Since the BPEL would be transformed to FIACRE, the verification would consists in analyzing the FIACRE code corresponding to these observers.

2. Encode the observers directly at the FIACRE level.

BPEL level observers We have defined an original idea of giving the observers at the BPEL level. The main advantage of this technique is that these observers may be used independently of both of the verification language and the BPEL editor.

Bounded Response Property The bounded response property could be written as $\Box(receive \Rightarrow \Diamond_{\leq T} finish)$ which means that `finish` must occur within T time units after the end of the receive activity. In order to verify such a property on the BPEL process, we need to measure the delay between the reception of a request and its corresponding reply. This verification is done on two levels :

1. Every `receive` activity is followed by a `reply` activity. It is the same as verifying a structural property.
2. The elapse of time between the receive activity and the end of the process is bounded by a fixed time T. This is achieved by defining a timed observer at the BPEL code level (Fig. 3.4).

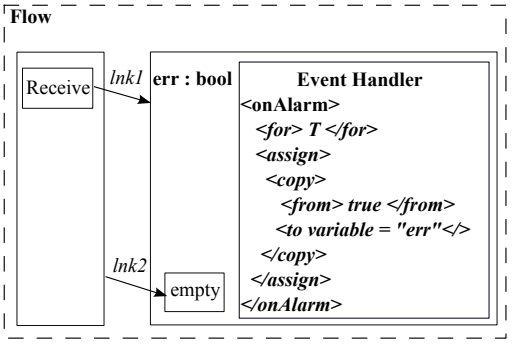


Figure 3.4 – *Bounded Response Observer*

The observed system is encapsulated inside of a flow where another scope that plays the role of the observer is added. This scope contains an event handler in which the duration of the constraint is fixed. Whenever this duration is reached, a variable `err` is valuated to True reflecting the violation of the property. Furthermore, the event handler should start at the moment of the reception of the first request. This is represented by adding a link (`lnk1`) from the receive activity to the scope of the event handler. Moreover, at the end of the observed activity, the event handler should be terminated. That is why another link (`lnk2`) is added from the observed activity to the primary activity of the scope (empty).

Maximal Duration Property The observer shown in Fig 3.7 measures the maximal duration of an activity. For this purpose, the idea is to englobe the activity in question in a sequence with an empty activity. The finish of the empty activity -which happens to be the start of the studied activity- will trigger the event handler by sending `lnk1`. The finish of the activity in question will then end the observation by signaling `lnk2`. An error is signaled in case the duration is reached before the arrival of `lnk2`.

Minimal Duration Property Conversely to the previous observer, the observer shown in Fig. 3.6 measures the minimal duration of an activity. This observer is similar to the one we have already seen. The only difference is that the `err` variable is initialized to True.

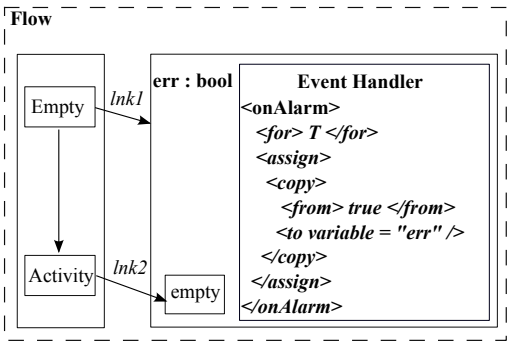


Figure 3.5 – *Maximal Duration Observer*

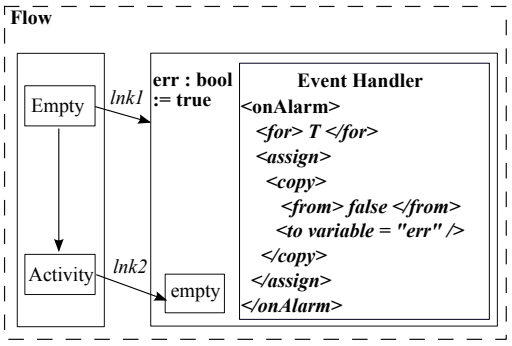


Figure 3.6 – *Minimal Duration Observer*

FIACRE level observers The second alternative is to build the observers at the FIACRE level. We show in Fig. 3.7 examples of observers that may be used to verify the described constraints :

1. The first observer checks whether the delay between two signals is bounded by a certain constant. The observer starts upon the reception of the event `e1`. An error is signaled only in the case the event `e2` takes more than `k` u.o.t to occur. This observer allows us to verify both of the Process Duration property and the difference between two activities property. Actually, the events `e1` and `e2` of the observer can be assumed to be respectively either the start and the finish of a single activity (Process Duration constraint) or the start of an activity and the finish of another (Difference between two activities constraints). However, in case the required events of the observer correspond to internal events of the system which means that they do not appear at the external level, then there is a need to navigate in the hierarchical structure of the component in order to reach these events. A solution is to transform these events into global events so they would appear in the interface of the system.

2. The second observer is used to verify the minimal duration property. The observer starts upon the reception of the event `e1`. An error is signaled only in the case the event `e2` is enabled before the minimal expected delay of `k` units of time. The same discussion concerning the case of internal events also holds here.

Finally, in all of these observers, we verify that the error state is always avoided. This could be written in LTL as $\Box\neg error$.

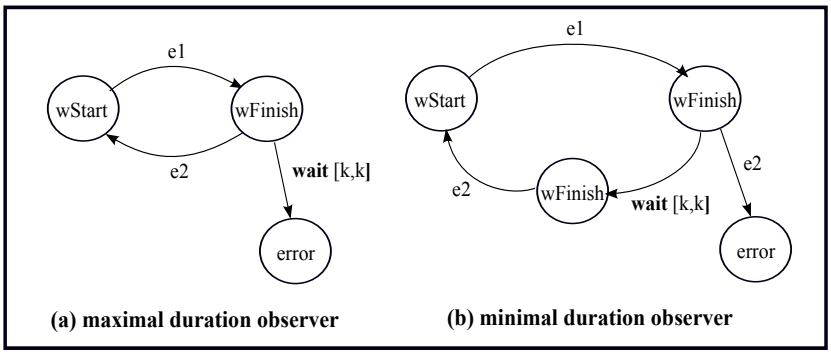


Figure 3.7 – *Observers at the FIACRE level*

3.4 Conclusion

In this chapter, we have shown what kind of properties may be verified in our framework. We have shown how an example of structural properties can be verified via an instrumentation of the FIACRE code. Interesting timed properties can also verified via timed observers which are either build at the BPEL or at the FIACRE level. In the next chapter, we see by means of a use case how the simulation can be integrated in the verification of BPEL processes.

Chapter Four

Use Case

4.1 Introduction

In this chapter, we consider a use case to illustrate our technique. The use case has two objectives : first, to demonstrate the use of the patterns by building its corresponding FIACRE code. Second, to show how refinement can be embedded in the transformation. We also prove some of the BPEL correctness properties on instances of the FIACRE patterns.

4.2 Use Case

We consider a timed variant of the purchase order example taken from the BPEL specification release [18].

In fact, upon receiving a purchase request, the process initiates simultaneously three sequences of activities which are used for calculating the final price, selecting a shipper, and scheduling the production and shipment for the order. Once the three tasks are completed, the process sends an invoice to the customer. However the final price cannot be calculated before the reception of the shipping price and the shipping date is needed for completing the scheduling task. This is represented respectively by a link from the `invoke shipping` activity to the `invoke`

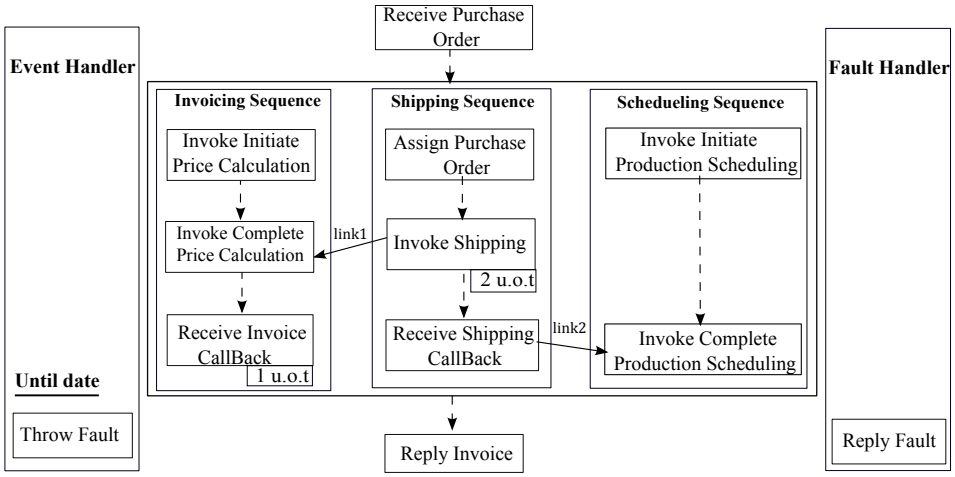


Figure 4.1 – *Timed purchase example*

`final price calculation` activity and a link from the `receive shipping callback` activity to the `invoke complete production scheduling` activity. However, a deadline inside of an event handler is fixed. After this deadline is reached, the event handler will proceed by signaling a fault meaning that the purchase instance is not treated. This means that the service is only offered until a certain date. Likewise, whenever a fault is thrown during the execution of the purchase sequence, the fault handler will terminate the process and reply a fault to the customer.

Finally, additional safety timed constraints are added to the use case. These constraints describe the requirements that need to be satisfied by the processes.

1. The total duration of the invoke activity `Invoke Shipping` takes 2 units of time (2 seconds).
2. An explicit wait of 1 u.o.t is added after the `Receive Invoice CallBack` activity to say the invoicing sequence takes 1 u.o.t.

4.2.1 Abstract Use Case Modeling and Verification

There exists two approaches to build a modular system. A Top/Down refinement-based approach where details are added progressively to the system and a Bottom/Up abstraction-based approach where details are abstracted progressively. BPEL follows a Top/Down approach but the language does not support the specification of abstract activities, i.e activities that allow a non-deterministic behavior. This is because BPEL is designed to be an implementation language and not a specification language. In our use case we thus follow a Bottom/Up approach : starting from the BPEL process, we choose to abstract some BPEL pieces and give their abstract specifications directly in FIACRE. The user properties are then verified on an abstract modeling of the use case containing these abstract specifications. Afterwards, we prove the simulation between these abstract specifications and their concrete implementations corresponding to the real BPEL code.

More precisely, for FH_a (FH_c) denoting the abstract (concrete) fault handler, P_a (P_c) denoting the abstract (concrete) purchase sequence and EH denoting the event handler what we do is the following :

$$((FH_a \| P_a) \| EH) \precsim_{DS} ((FH_c \| P_c) \| EH) \qquad \textit{Process Simulation}$$

In order to prove this simulation we proceed by verifying the following two simulation relations :

$$FH_a \| P_a \precsim_{DS} FH_a \| P_c \qquad \textit{Purchase Simulation}$$

$$FH_a \precsim_{DS} FH_c \qquad \textit{FaultHandler Simulation}$$

Here we note that in the *Purchase Simulation* clause, the FH_a is added because P and FH communicate via a shared variable *stop* that is put to true whenever a stop is signalled by FH.

Abstract Activity

We consider an abstract view (Fig 4.2) of the flow pattern of the purchase example. This means, that the flow activity will be considered as a black box. The only thing we know is that it behaves as a normal activity, it starts, it finishes and between its start and finish, it may input or output messages, faults or stops. Furthermore, we say that the duration of this activity is not less than 1 u.o.t, meaning that the difference between the start and the finish of the abstract activity is not less than 1 u.o.t.. This corresponds to the time the activity remains in state s1 and is modeled in FIACRE in a way that the events `fault`, `O` and `I` do not reinitialize the clock of the event `treat`. The events highlighted in Fig 4.2 are local events.

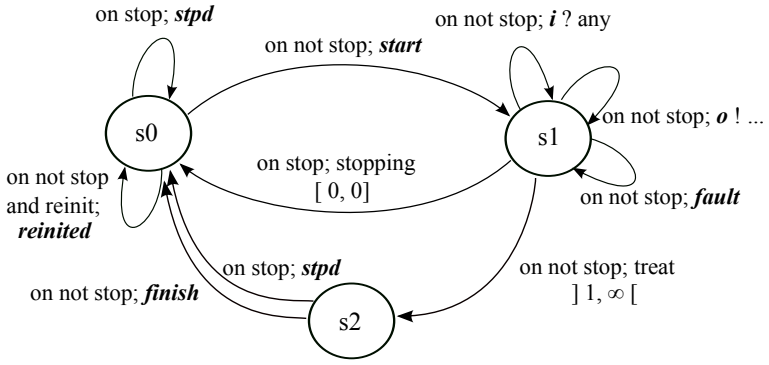


Figure 4.2 – *Abstract Activity in FIACRE*

Abstract Fault Handler

We consider an abstract view (Fig 4.3) of the fault handler of the purchase example. What we know is that upon receiving a fault, the fault handler starts a stopping mechanism to stop the activities of the process. In our example, the fault handler proceeds by executing a reply activity [1].

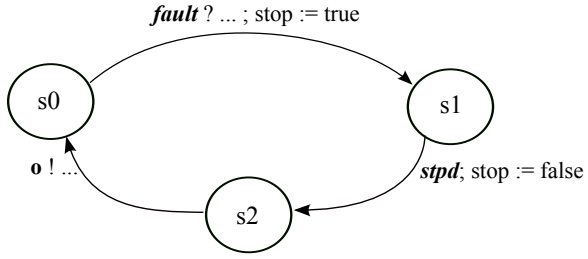


Figure 4.3 – *Abstract Fault Handler in FIACRE*

[1]Here we assume that this reply activity does not throw any faults. Otherwise a fault thrown inside of the fault handler's reply activity should be added so that it terminates the whole BPEL process instance. But here the reply activity is the last activity to execute and the process will terminate anyway.

Abstract Use Case Modeling

We give in the following the root component of the FIACRE code associated with the abstract case study. In this excerpt, only the useful ports and shared variables are shown.

```
component PurchaseProcess [I: inputs, O: outputs] is
port S,F,after,stpd: none,
     fa: fault
var stop: bool= false,
    link: t_link,
    vars: t_var
par /* BPEL process pattern with timer */
    after,faIN,stpd →
      process_act [S,F,I,O,fa,stpd,after](& link,&stop,&var)
 || faIN,stpd,exit →
      fault_handler_a [fa,O,stpd](&link, &stop, &var)
 || after → timer [after]
end
```

On one hand, we compose the primary activity of the process which contains the `purchase` sequence (and thus the abstract activity) and the event handler and on the other the abstract fault handler. Additionally, the `Timer` process runs concurrently and synchronizes on the `after` port with the system. The `process_act` may generate faults by synchronizing with the fault handler component on the `Fau` port. This triggers a termination mechanism done by communications through the shared variable `stop` and the port `stpd`.

In Fig 4.4 we give in greater details the structure of the FIACRE code. This structure consists of two interacting components. The `Purchase` component containing the purchase sequence, the event handler component and the `Fault_H` component. These two communicate via the `Fau` event with which the purchase sequence and the event handler signal their internal faults. The purchase sequence contains three activities that run in sequence : each of the activities communicate its start signal with the finish signal of the one preceding it in the sequence (`con_2_rec`, `rec_2_as`,`as_2_inv`, `rep_2_con`). The event handler mainly contains an alarm controller that synchronizes on a `after` event. This event is communicated in a non-deterministic manner by a timer process that signals the reaching of the absolute time. In this case, a throw activity is run by communicating a fault to the `Fault_H_a` component. In response, the fault handler initiates a stopping mechanism by communicating respectively with the `stop` shared variable and the `stpd` event followed by a reply activity.

Use Case Verification

Now, properties may be verified on the underlying code. We will show examples of properties that may be verified in this use case. We will start by verifying the correctness properties. Then we will check a user requirement property namely, the (bounded) response property in the timed and the untimed contexts. For verification purposes, the system needs to be closed, so we model an environment process *Environment* as an input/output enabled process that runs concurrently with the purchase pro-

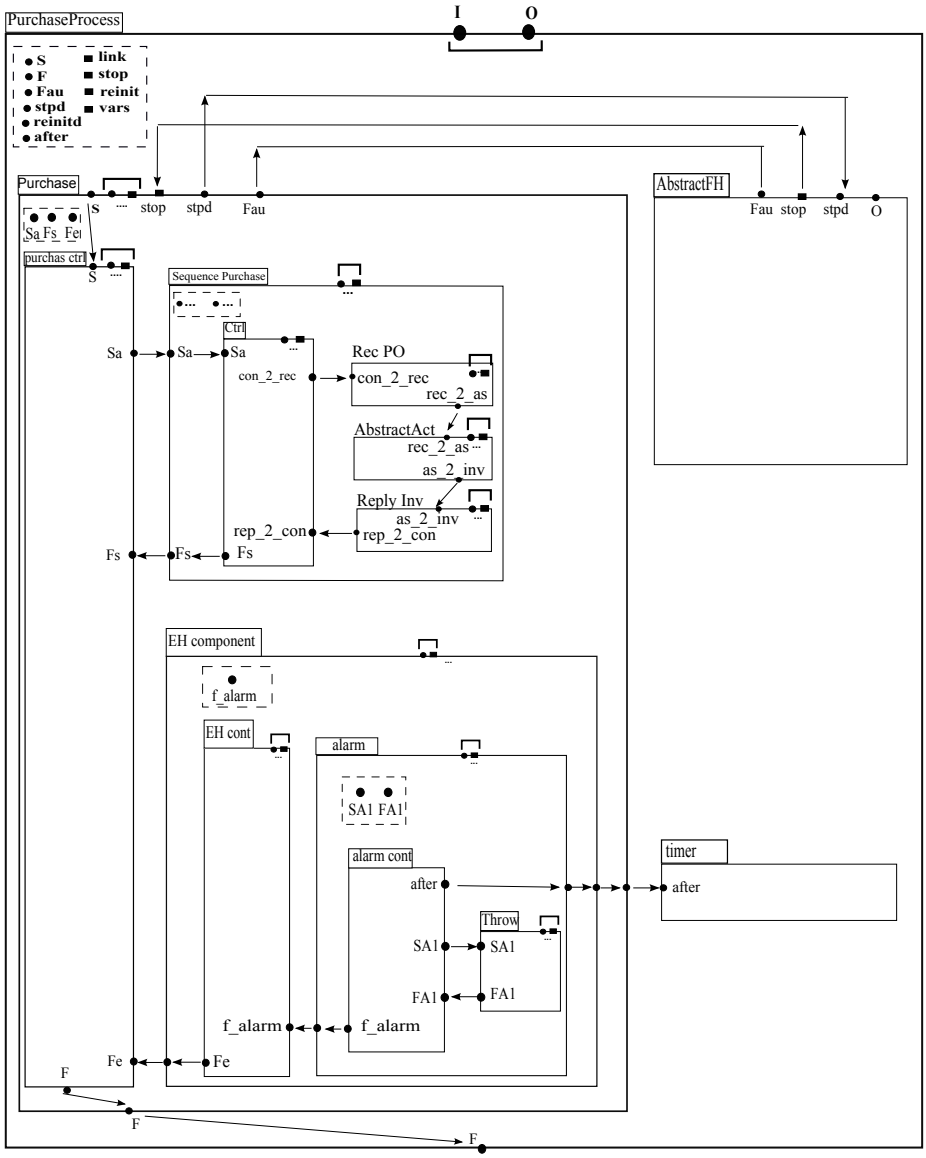


Figure 4.4 – *Modeling of the Abstract Use Case in FIACRE*

cess. The verification is then made on the completely synchronous composition :

$$ClosedPurchaseProcess = Environment \| PurchaseProcess$$

Correctness Properties At this level we check the correctness of the event handler (Fig 4.4).

Unlike what we have seen in the patterns, the start of the event handler depends on the finish of the initial start activity which corresponds here to the receive purchase activity.

$$ClosedPurProc \models \Box(\#startEH \Leftrightarrow \#finishReceivePurchase)$$

$startEH$ and $finishReceivePurchase$ are the aliases of respectively the event handler transition that starts the event handler and the last transition of the receive activity.

Now we verify the two event handler disablement properties. Intuitively, we verify by the first property that the purchase process may not finish unless the finish signals of the event handler and the purchase sequence are received.

$$ClosedPurProc \models (\neg F \ \mathbf{W} \ Fs) \wedge (\neg F \ \mathbf{W} \ Fe)$$

The second property means that if the purchase sequence and the event handler finish in whatever order then the purchase process eventually either finishes or a fault is thrown.

$$ClosedPurProc \models \Box(((Fs \wedge \Diamond Fe) \vee (Fe \wedge \Diamond Fs)) \Rightarrow \Diamond(F \vee Fau))$$

User-Requierements Properties : Temporal Verification Starting with the untimed context, the following *response* property guarantees that independently of the starting date of the system, a purchase demand is always followed by a reply. This reply can either be a positive reply in case of a successful purchase or a negative reply in case the timing constraint of the event handler was violated or simply in case a fault was thrown during the treatment of the command.

$$ClosedPurProc \models \Box(\ purchase\_demand \Rightarrow \Diamond(Reply\_Invoice \vee Reply\_Fault))$$

However the verification of either one of the following properties is not satisfied. This is because, neither one of the two replies is always verified.

$$ClosedPurProc \not\models \Box(\ purchase\_demand \Rightarrow \Diamond(Reply\_Invoice))$$

This property allows us to say that our use case does not ignore the faults.

$$ClosedPurProc \not\models \Box(\ purchase\_demand \Rightarrow \Diamond(Reply\_Fault))$$

As for this property, it means that our use case may finish successfully.

User-Requierements Properties : Timed Verification As we have said earlier, the verification of timed properties is made by using timed observers. We give in the following code the observers for the verification of the total duration of the process property.

```
process checkProcessDuration[start: none,finish: none] is
  states s0,s1,s2,error
  init to s0
from s0
  start;to s1
from s1
  select
   finish;to error
□  wait ]1,...[;to s2
  end
from s2 finish ;to s0
```

The minimal duration of the purchase is checked via an observer measuring the time between the start and finish of the process. The observer is enabled once the purchase process is started. This is done via a synchronization with the start event event of the process (code line 5). Afterwards, in state s_1, the observer waits for either a finish signal to arrive (code line 8) which is synchrnonized with the finish of the purchase process or for the time constraint to be fullfiled (code line 9). Now in case the finish signal arrives before the minimal duration of 1 units of time, an error is signaled by transiting towards the *error* state. This means that the purchase process has executed in less than 1 u.o.t and thus violates the time constraint.

Now, once the observer is composed with the actual system, we can proceed by the verification of their attached properties. These properties consists in verifying that an observer never reaches its corresponding error states. This means that the transition that leads to the `error` state is never taken. This would also mean that the properties are satisfied.

4.2.2 Concrete Modeling

The concrete system that implements the abstract activity consists in the result of the transformation of the BPEL flow activity of the purchase example. This transformation results in the FIACRE component shown in Fig 4.5. The flow controller signals concurrently the start of three sequences by synchronizing on the `Sa` event. It then awaits for their respective finish signals (`Fa1`, `Fa2` and `Fa3`). The three sequences correspond to the invoicing, shipping and scheduling sequence of Fig 4.1.

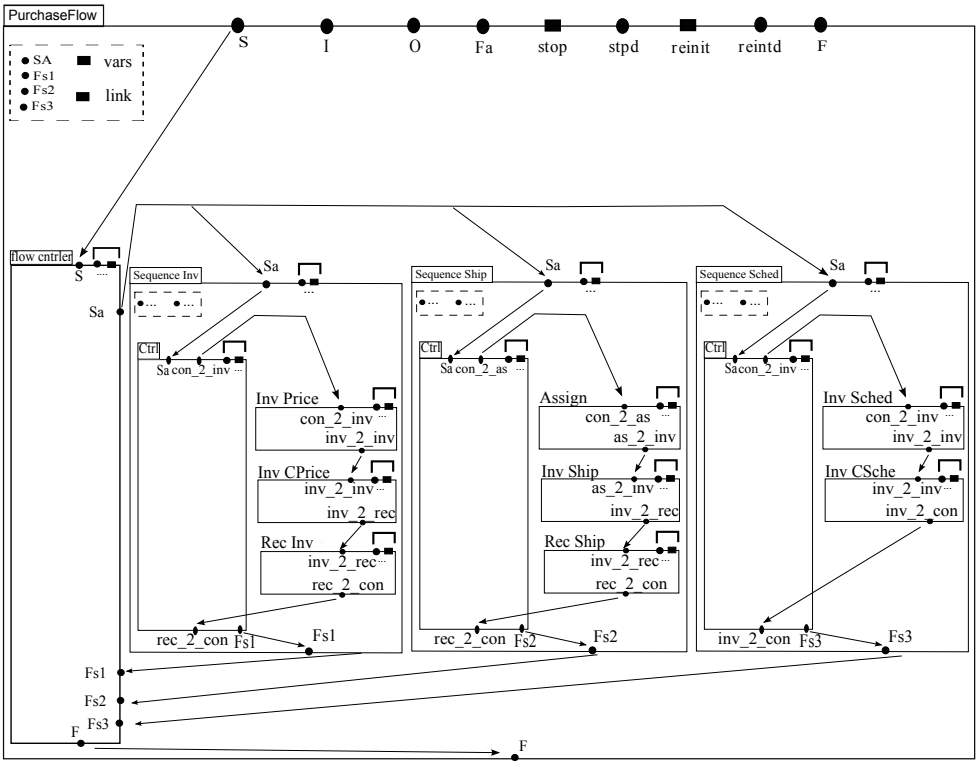


Figure 4.5 – *Modeling of the Concrete : Purchase Flow*

As for the concrete fault handler, it consists in the transformation of the

Fault handler of the purchase example. Upon the reception of a fault, the fault handler stops the process and then executes the reply fault activity. This is shown in Fig 4.6.

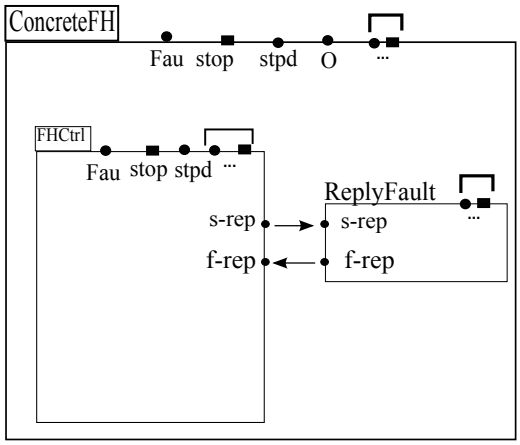


Figure 4.6 – *Modeling of the Concrete Fault Handler*

Verification of Additional Correctness Properties At this level we check the correctness of the sequence, flow, BPEL links and fault handler. The verification is made on the completely synchronous composition of the *PurchaseFlow* (Fig 4.5), *concreteFH* (Fig 4.6) and an input/output enabled environment process.

$$Concrete = Environment \| PurchaseFlow \| ConcreteFH$$

We start by checking the correctness of the succession of the start and finish events of the subactivities of the sequence pattern. We consider the invoicing sequence :

The initial price calculation invoke does not start if the invoicing sequence is not started :

$$Concrete \models \neg con\_2\_inv \ \mathbf{W} \ Sa$$

Neither the invoicing sequence finishes if the invoice `receive` activity is not finished :

$$Concrete \models \neg Fs1 \ \mathbf{W} \ rec\_2\_con$$

Finally, the finish signal of invoice `receive` activity will either lead to the sending of the invoicing sequence finish signal or to the sending of the stpd signal.

$$Concrete \models \Box(rec\_2\_con \Rightarrow \Diamond(Fs1 \vee stpd))$$

A flow is not started before the start of its embedded activities.

$$Concrete \models \neg Sa \ \mathbf{W} \ S$$

A flow may not finish without receiving the finish signals of all of its embedded activities. This is verified by the following three properties.

$$Concrete \models \neg F \ \mathbf{W} \ Fs1$$

$$Concrete \models \neg F \ \mathbf{W} \ Fs2$$

$$Concrete \models \neg F \ \mathbf{W} \ Fs3$$

Now, in order to check the BPEL links correctness property we consider the complete price calculation invoke activity which accepts an incoming link. Intuitively, we want to verify that the start signal of the complete price calculation invoke activity may not occur unless the activity has received all its incoming links. This is verified by the following property :

$$Concrete \models \Box(inv\_2\_inv \Rightarrow link.InvCPrice.counter = 0)$$

Finally, a weak termination is verified. This means that the fault handler behaves correctly.

$$Concrete \models \Box(stop \Rightarrow \Diamond stpd)$$

4.2.3 Proving the Refinement

In order to be able to conclude that the user-requierements properties (temporal and timed) verified at the abstract level also hold at the concrete level, we need to verify that a simulation holds between the asbtract specifications and their implementations. We shall then proceed by proving the refinement between these two.

Abstract and Concrete Activities Simulation

First we start by proving the following simulation :

$$AbstracFH \| AbstractAct \precsim_{DS} AbstractFH \| PurchaseFlow$$

Following our technique, two steps need to be done. First to compose the systems with the control and time observers and second to verify the satisfaction of the μ-calculus property on the composition.

Composition with the Observers The abstract and the concrete systems are composed with the two observers after having renamed each of their interface events with the corresponding suffix (_c for the concrete events and _a for the abstract events) and their local events are made global (the events of the dashed boxes are transformed to interface events). The names of the local events of the abstract and concrete systems are left unchanged. Actually it is not important whether they are changed or not since their names are not shared between the abstract and the concrete system. The systems are thus composed in the following way (Fig 4.7) :

$$(AbstractFH \| AbstractAct) \ ||| \ (AbstractFH \| PurchaseFlow) \| Obs \| Obs_{\delta}$$

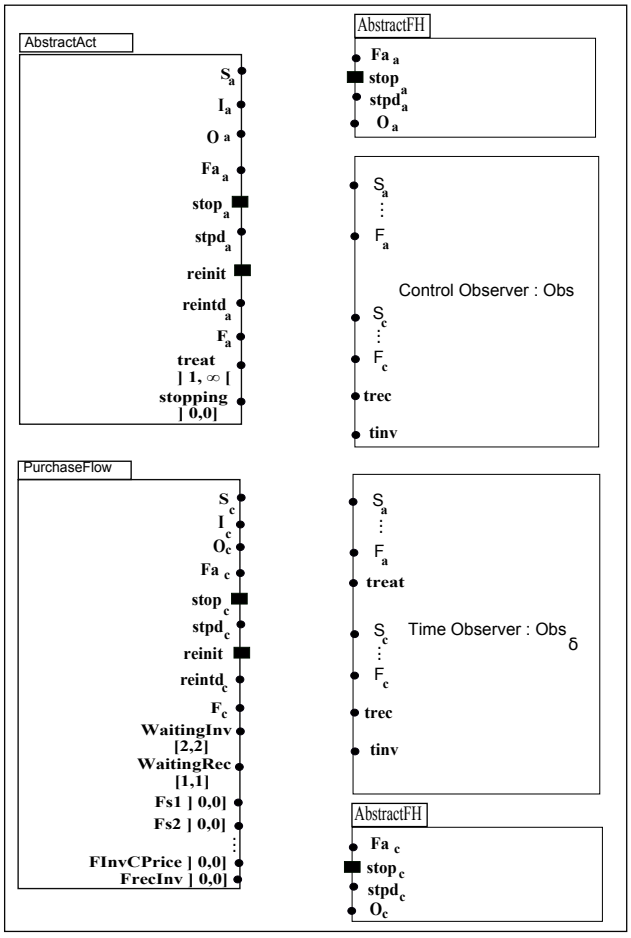


Figure 4.7 – *Composition with the Observers*

Simulation Verification The composition process is model checked by the following state-event property that preserves the deadlock :

$$(q_a^0, q_c^0, ok, evt0) \models_{DSTimedWeakSimulation}$$

$$\overbrace{\nu X.Obs\ in\ ok \wedge Obs_\delta\ in\ evt0 \wedge \bigwedge_i [e_c^i](\mathbf{EF}_{\tau_a}\langle e_a^i\rangle X) \wedge \bigwedge_j [\tau_c^j] X \wedge}$$

$$(\mathbf{EF}_{\tau_a}\langle delay\rangle \top) \Rightarrow \mathbf{EF}_{\tau_a}(\langle delay\rangle \top \wedge [delay]\mathbf{EF}_{\tau_a} X) \wedge$$

$$\bigwedge_i [e_a^i]\langle e_c^i\rangle (stop_c = stop_a \wedge reinit_c = reinit_a) \wedge$$

$$\overbrace{(stop_c = stop_a \wedge reinit_c = reinit_a)}^{PropositionsRelation}$$

where

$$e_c^i = \{S_c, I_c, O_c, FA_c, stpd_c, reinited_c, F_c\}$$

$$e_a^i = \{S_a, I_a, O_a, FA_a, stpd_a, reinited_a, F_a\}$$

$$\tau_a = \{treat, stopping\}$$

$$\tau_c = \{SAll, FS1, FS2, FS3, SRecInv, SInvPrice, SInvCPrice, WaitingInv, SRecShip, WaitingRec, SInvShip, FRecShip, noWait, SInvSch, SInvCSch, FInvCSch\}$$

The formula is a direct instantiation of the DS-Timed Weak Simulation property pattern we have given in Chapter : Timed Weak Simulation Verification. The set of states returned by the μ-calculus property contain the initial state of the process, the simulation is then verified. Now suppose that the time interval of `treat` in the abstract system is changed to $]1, 2]$. In this case, the μ-calculus property does not hold. This is because the concrete system violates the time allowed (more than 2 u.t.) by the specification.

Fault Handler Simulation

We are left with verifying the following simulation :

$$AbstracFH \precsim_{DS} ConcreteFH$$

For this, the same verification technique is applied for the case of the fault handler and the simulation test on the composition $AbstractFH \;|||\; ConcreteFH \| Obs \| Obs_{\delta}$ also holds.

Finally, since in both of the simulation tests a deadlock preserving simulation is satisfied then the properties verified on the abstract use case are preserved by the concrete system.

4.3 Conclusion

In this use case, we have been able to accomplish two points. First, we have shown how the BPEL correctness properties are satisfied on instances of our FIACRE patterns. Second, we have shown how abstraction-based model checking and simulation verification can be combined to verify BPEL models. We started by considering abstract models of BPEL. Then, we verified user properties on the abstract use case. Finally, we finished by showing that a simulation actually holds between the abstract models and their concrete implementations.

Part IV

Conclusion and Perspectives

Chapter One

Conclusion and Perspectives

1.1 Conclusion

The context of this PhD is the FIACRE language and more precisely the verification, via model checking, of web services using the FIACRE language. In order to ease this verification and trespass the shortcomings of model checking, we were led to the notion of refinement of timed systems in FIACRE. The idea is to integrate the refinement of timed systems in the verification chain of web services (BPEL). Having this in mind, we would need a refinement definition that allows to guarantee the preservation of properties all along the system development. We also need it to be compositional, in the sense that replacing a component by another that refines it would not affect the whole composition, i.e the global properties of the composition remain preserved. A model checking verification chain for BPEL could then be followed via a transformation from BPEL to FIACRE.

In this PhD document, we have achieved the above-stated objective by suggesting the following contributions:

1. Definition of the timed model (CTTS) that captures the specificity of FIACRE. We have shown as well that the defined model satisfies the compositionality property.

2. Definition of a timed simulation relation. We have shown that the adopted definition guarantees the important properties of trace inclusions and thus preserve linear properties. We have also proven that our simulation is compositional.

3. A technique to verify the timed simulation between timed transitions systems originating from CTTS. For this, we have followed a standard technique in model checking –a timed observer and an untimed logic property– that is mostly used in the verification of timed properties. The idea is in fact based on an observer with which A and C are composed. The result of their composition is then analyzed using a μ-calculus property that captures the timed weak simulation definition. One important advantage of our approach is that it is self-contained and relies exclusively on existing model checking tools, that is, no specific algorithm for the simulation verification is given. Moreover, some of the restrictions that exist in the literature

have been relaxed, namely the silent actions in the abstract system. However, since our verification technique is a model checking based one, its main disadvantage remains the same as that of model checking which is the state explosion for large systems.

4. We have shown that for a given class of systems, the μ-calculus criterion used for the simulation verification is sound and complete.

5. Definition of a transformation from WS-BPEL 2.0 to the FIACRE specification language : the transformation is mostly structural and is based on a set of patterns covering both behavioral and timed aspects of BPEL. Moreover, an interesting feature of this transformation is that the hierarchical composition of the BPEL activities is preserved in FIACRE. This comes from the fact that FIACRE is a component-based language which the same as BPELŠs. Another interesting feature is that we have shown how a subset of the FIACRE patterns may be proven correct w.r.t their BPEL semantics. This work may be extended to all of the BPEL constructs of the transformation.

6. A Verification Framework that supports a rich set of properties: the most noted advantage of our transformation to FIACRE compared to the literature is the handling of time properties. These properties could be written in MITL and are handled here using timed observers which are defined at either the BPEL or the FIACRE level. The specification of observers at the BPEL level remains a novel idea and may be a starting point for further studies.

7. A BPEL use case that illustrates our work and shows how abstraction-based model checking and simulation verification can be combined to verify BPEL models (and more generally component based models).

1.2 Perspectives

We move now into discussing the perspectives of the work presented in this document. Actually, this work opens the door to several key points, in terms of optimization, of complementary studies, and even in terms of new ideas to tackle and study. In terms of optimization, the studies could be followed in the direction of relaxing or at best removing the hypothesis made on the systems in the simulation verification technique. In terms of complementary studies, the idea of refining FIACRE processes corresponding to BPEL activities can be extended to the refinement of a whole web service. This leads to the possibility of substituting a web service by another in a web service composition. Also in terms of complementary tasks, this work may strongly be considered for implementation. Finally, in terms of new ideas to tackle, run time verification of BPEL is an interesting idea that may be subjected to further studies and research by means of BPEL-level observers.

1.2.1 Simulation Verification Technique

The simulation verification technique may be approached using two independent techniques. In the first technique, we either relax the $1 - \tau$ or we develop a dedicated algorithm as done in the ECDAR tool [1] or in the Vesta tool [66].

The purpose of relaxing the main hypothesis of $1 - \tau$ in the abstract system is to accept multiple successive silent actions τ. If this hypothesis is removed, it will allow more space for design and will ease the progressive development of systems. This is particularly true because a concrete system at the previous level could then be used directly as an abstract system at the next level.

Developing an algorithm may reduce the risks of state explosion of model checking. However, we still need to investigate its cost, complexity, and the number of hypothesis that may thus arise.

1.2.2 FIACRE Extensions

FIACRE Extension with composition operators at the components level : actually, in FIACRE apart from the parallel composition which is done at the components level, the sequence, the choice and other basic operations are all made at the process level. Extending FIACRE with component level composition operators would ease the transformation from BPEL to FIACRE. Most of the start/finish signals in the FIACRE patterns would disappear. Moreover, a component level termination operator could be easily applied without needing to prefix all the transitions by stop variables. For example, for C_1 and C_2 two FIACRE components, the syntax could be extended as follows :

$$C_1[...] \textit{ terminate on guard } \| \; C_2[...]$$

The semantics of this operator reflects a termination request for the components embedded in C_1 whenever the component C_2 evaluates the guard to True.

Adding such operators would make the transformation from BPEL to FIACRE more reusable.

1.2.3 Run Time Verification of BPEL

Define a wider range of BPEL observers and extend their study in order to allow the run time verification of BPEL : The idea is to propose a library of observers used by the developers in order to instrument their code for verification purposes. The observers will thus run at the same time of the real BPEL code and would monitor (or control) the execution of the application. The observation results can be sent to log files in order to save the execution trace of the BPEL code. Other associated actions are for instance to halt the service in case of occurrence of undesired events etc.

1.2.4 Tool Chain for the Simulation/ Verification of BPEL

Developing tools for the simulation between timed systems at one hand, and the verification of BPEL at the other, can offer several tasks. The first

task is the interactive selection of the BPEL piece to abstract. This is followed by an automatic generation of the abstract and concrete FIACRE code of the selected piece. Now, in order to prove the refinement between the abstract and concrete system without needing to manipulate complex logical formulas, the tool should automatically generate the μ-calculus simulation property to be verified. Finally, for verification purposes, the tool may offer an interactive selection of property patterns and observers used for model checking or for run time verification.

Part V

Appendix

Chapter One

Appendix A : Building Systems with Label Structures

1.1 Label Structure

We start by describing the transition systems labels by means of a label structure.

Definition 1.1 (Label Structure) *A label structure is a tuple $\langle L, \bowtie \rangle$ where L is a set of labels and $(\bowtie : L \times L \nrightarrow L)$ is a partial binary composition operator over L.*

The function is partial because some composition may be blocked since $\bowtie$ describes exclusively synchronous compositions. The asynchronous aspects are covered later (see LTS composition). Our composition then models:

1. A successful synchronization between l and l' that results in $l \bowtie l' \in L$.
2. A blocking synchronization between l and $l' : (l, l') \notin \mathbf{dom}(\bowtie)$.

Let the reader not confuse our label structure with other event structuring propositions, namely with the event structures [103]. The event structures models the occurrence of events during the system *execution* via an introduction of a causal dependency relation and a conflict relation between the events. While in our case, we introduce a label structure which models the way the labels (i.e events) are *statically* composed.

Definition 1.2 (Associativity of a Label Structure) *Given a label structure $LS = \langle L, \bowtie \rangle$, LS is said to be associative if its composition operator $\bowtie$ is associative. Formally, for l_1, l_2 and $l_3 \in L$, LS is associative if :*

1. $(l_1, l_2) \in \mathbf{dom}(\bowtie) \wedge ((l_1 \bowtie l_2), l_3) \in \mathbf{dom}(\bowtie) \Leftrightarrow (l_2, l_3) \in \mathbf{dom}(\bowtie) \wedge (l_1, (l_2 \bowtie l_3)) \in \mathbf{dom}(\bowtie)$.
2. $(l_1, l_2) \in \mathbf{dom}(\bowtie) \wedge ((l_1 \bowtie l_2), l_3) \in \mathbf{dom}(\bowtie) \Rightarrow ((l_1 \bowtie l_2) \bowtie l_3) = (l_1 \bowtie (l_2 \bowtie l_3))$.

1.1.1 Examples of Label Structures

Time Label Structure. For Δ a time domain (non negative real numbers, naturals ...) equipped with a binary associative operator $+$ and a neutral

element 0, the time structure TS is given in the following where a synchronization only takes place when the same value is present at the two sides of the composition :

$$TS = \langle \Delta, (\delta_1, \delta_2) \mapsto \delta_1 \textbf{ if } \delta_1 = \delta_2 \rangle$$

Basic CSP Synchronizing Events Structure. Here, we model the case of the completely synchronous composition of CSP. For C a set of communication ports, a synchronizing structure on C is the label structure :

$$Sync_{CSP} = \langle C, (c_1, c_2) \mapsto c_1 \textbf{ if } c_1 = c_2 \rangle$$

CCS Synchronizing Events Structure. For C a set of events and $C? = \{c? \mid c \in C\}$ and $C! = \{c! \mid c \in C\}$, this is represented in our label structure as follows :

$$Sync_{CCS} = \langle C? \cup C! \cup \{\tau\}, (c!, c? \mapsto \tau) \rangle$$

1.1.2 Composition of Label Structures

We define the product and the sum of two label structures. The product operation builds new labels as couples of the composed labels. For example, this is used when composing synchronization and memory access labels. Unlike the product operation, the labels of the sum operation are defined over the union of the composed labels. This is used when composing synchronization and time labels to specify that only one of the events may occur but not both simultaneously.

Product of Label Structures

Given two label structures $\langle L, \bowtie \rangle$ and $\langle L', \bowtie' \rangle$, their product ranges over the set $P = (L \cup \{\epsilon\}) \times (L' \cup \{\epsilon'\}) \backslash \{(\epsilon, \epsilon')\}$ where ϵ (resp. ϵ') is a new element of L (resp. L') supposed to be neutral for the $\bowtie$ operator of its respective label structure. For l_1, $l_2 \in L$ and l_1', $l_2' \in L'$, the composition of $(l_1, l_1'), (l_2, l_2')$ is defined only if the composition of l_1 and l_2 and the composition l_1' and l_2' are also both defined.

$$\langle L, \bowtie \rangle \otimes \langle L', \bowtie' \rangle =$$
$$\langle P, (\ \ (l_1, l_1'), (l_2, l_2') \mapsto (l_1 \bowtie l_2, l_1' \bowtie' l_2') \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie) \wedge (l_1', l_2') \in \textbf{dom}(\bowtie')\ \ ) \rangle$$

Sum of Label Structure

Given $\langle L, \bowtie \rangle$ and $\langle L', \bowtie' \rangle$, their sum ranges over the disjoint union of $L^\bullet \uplus {}^\bullet L'$ where $L^\bullet = \{l^\bullet \mid l \in L\}$ and ${}^\bullet L' = \{{}^\bullet l \mid l \in L'\}$.

$$\langle L, \bowtie \rangle \oplus \langle L', \bowtie' \rangle = \langle L \uplus L', \begin{pmatrix} l_1{}^\bullet, l_2{}^\bullet \mapsto (l_1 \bowtie l_2)^\bullet \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie) \\ {}^\bullet l_1, {}^\bullet l_2 \mapsto {}^\bullet(l_1 \bowtie' l_2) \textbf{ if } (l_1, l_2) \in \textbf{dom}(\bowtie') \end{pmatrix} \rangle$$

Proposition 1 (Preservation of Associativity) *Given the label structures LS and LS′, $LS \oplus LS'$ (resp. $LS \otimes LS'$) is associative if LS and LS′ are associative.*

Proof. The proof is based on checking $\forall l_1, l_2, l_3 \in LS \oplus LS'$ (resp. $LS \otimes LS'$) whether $(l_1 \bowtie l_2) \bowtie l_3$ and $l_1 \bowtie (l_2 \bowtie l_3)$ are defined simultaneously and have the same value. It has been checked using the Coq [102] assistant prover and is based on an automated case analysis. For instance, in the $LS \otimes LS'$ case, this check induces 127 cases in which 83 are non trivial. □

1.2 Labeled Transition Systems LTS

Definition 1.3 (Labeled Transition System LTS) *Given $LS = \langle L, \bowtie \rangle$, a labeled transition systems $\mathcal{L}$ over LS -denoted as $\mathcal{L}_{LS}$- is defined as $\langle Q, Q^0 \subseteq Q, T, \alpha : T \to Q, \beta : T \to Q, \lambda : T \to L \rangle$ where Q, Q^0, T denote respectively the sets of states, initial states, transitions and α, β and λ denote functions that return respectively, the source, the target and the label of a transition.*
We write $q \xrightarrow{l} q'$, when there exists $t \in T$ such as $\alpha(t) = q$, $\beta(t) = q'$ and $\lambda(t) = l$. Furthermore, we define the alphabet of an $\mathcal{L}_{LS}$ -denoted as $alphabet(\mathcal{L}_{LS})$- as the set of labels that are actually used by the transitions of $\mathcal{L}_{LS}$.

LTS Composition

Given two LTS defined over $\langle L, \bowtie \rangle$, a set of labels $S \subseteq L$ denoting the allowed synchronization results, the label composition function $\bowtie$ is extended to an LTS composition function $\langle \underset{S}{\bowtie} \rangle$ as follows:

$$\langle Q_1, Q_1^0, T_1, \alpha_1, \beta_1, \lambda_1 \rangle \langle \underset{S}{\bowtie} \rangle \langle Q_2, Q_2^0, T_2, \alpha_2, \beta_2, \lambda_2 \rangle = \langle Q_1 \times Q_2, Q_1^0 \times Q_2^0, T, \alpha, \beta, \lambda \rangle$$

where the set of transitions $T = \{t_1 \odot t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2\} \cup \{t_1 \uparrow_1 \mid t_1 \in T_1\} \cup \{t_2 \uparrow_2 \mid t_2 \in T_2\}$, α, β and λ are defined by the following rules :

$$\frac{t_1 : q_1 \xrightarrow{l_1} q_1',\ t_2 : q_2 \xrightarrow{l_2} q_2',\ (l_j, l_k) \in \mathbf{dom}(\bowtie) \wedge (l_j \bowtie l_k) \in S, \mathrm{j,k} \in \{1,2\}, \mathrm{j} \neq \mathrm{k}}{t_1 \odot t_2 : (q_1, q_2) \xrightarrow{l_j \bowtie l_k} (q_1', q_2')} \textsc{Synchronous}$$

$$\frac{t_1 : q_1 \xrightarrow{l_1} q_1' \ ,\ l_1 \notin S}{t_1 \uparrow_1 : (q_1, q_2) \xrightarrow{l_1} (q_1', q_2)} \textsc{Interleaving}_L \qquad \frac{t_2 : q_2 \xrightarrow{l_2} q_2' \ ,\ l_2 \notin S}{t_2 \uparrow_2 : (q_1, q_2) \xrightarrow{l_2} (q_1, q_2')} \textsc{Interleaving}_R$$

S is omitted when it is equal to L. In this case $\langle \bowtie \rangle$ is a fully synchronous composition operator. The rule `Synchronous` is made symmetric in order to maintain the commutativity of the LTS composition even if $\bowtie$ is not commutative.

Proposition 2 (Commutativity of $\langle \underset{S}{\bowtie} \rangle$) *$\langle \underset{S}{\bowtie} \rangle$ is commutative thanks to the two symmetric synchronous rules in the LTS composition.*

Proposition 3 (Associativity of $\langle \underset{S}{\bowtie} \rangle$) *Given $LS = \langle L, \bowtie \rangle$, the label sets $S_1, S_2 \subseteq L$ and the LTSs $\mathcal{L}_{1_{LS}}, \mathcal{L}_{2_{LS}}$ and $\mathcal{L}_{3_{LS}}$, $\mathcal{L}_1 \langle \underset{S_1}{\bowtie} \rangle (\mathcal{L}_2 \langle \underset{S_2}{\bowtie} \rangle \mathcal{L}_3) \simeq (\mathcal{L}_1 \langle \underset{S_1}{\bowtie} \rangle \mathcal{L}_2) \langle \underset{S_2}{\bowtie} \rangle \mathcal{L}_3$ if the underlying label structure LS is associative and commutative, S_1 and S_2 are stables over LS, $S_1 \subseteq S_2$, $alphabet(\mathcal{L}_1) \subseteq S_1$ and $alphabet(\mathcal{L}_2) \subseteq S_2$.*

This proposition is close to the weak associativity theorem of the CSP generalized parallel operator where this theorem is tackled by assuming that $S_1 = S_2$ [96]. Here we consider a variant of this theorem, where the hypothesis is generalized to $S_1 \subseteq S_2$. However, other hypothesis were added, one of them is the commutativity of the composition. Off course, in CSP the commutativity of the composition is always true since the composition is made between two identical events.

Proof. Given an associative label structure LS, two stable sets $S_1, S_2 \subseteq L$ over LS where $S_1 \subseteq S_2$, the LTSs $\mathcal{L}_{1_{LS}}, \mathcal{L}_{2_{LS}}$ and $\mathcal{L}_{3_{LS}}$ such as $alphabet(\mathcal{L}_1) \subseteq S_1$ and $alphabet(\mathcal{L}_2) \subseteq S_2$, we need to prove that $\mathcal{L}_1 \langle \underset{S_1}{\bowtie} \rangle (\mathcal{L}_2 \langle \underset{S_2}{\bowtie} \rangle \mathcal{L}_3) \simeq_R (\mathcal{L}_1 \langle \underset{S_1}{\bowtie} \rangle \mathcal{L}_2) \langle \underset{S_2}{\bowtie} \rangle \mathcal{L}_3$ where $R = \{(q_1, ((q_2, q_3)), ((q_1, q_2), q_3) \mid q_1 \in Q_1 \wedge q_2 \in Q_2, q_3 \in Q_3\}$. The idea consists in showing that each transition of one can be found in the other. Let $t : (q_1, (q_2, q_3)) \xrightarrow{l} (q'_1, (q'_2, q'_3))$ be a transition of $\mathcal{L}_1 \langle \bowtie \rangle (\mathcal{L}_2 \langle \bowtie \rangle \mathcal{L}_3)$. Two cases need to be investigated which are whether $l \in S_1$ or not.

1. Suppose $l \in S_1$: by unfolding the synchronizing rule we would obtain the two transitions $q_1 \xrightarrow{l_1} q'_1$ and $(q_2, q_3) \xrightarrow{l_{23}} (q'_2, q'_3)$ where $l = l_1 \bowtie l_{23}$ (by $\bowtie$ commutativity) and $l_1, l_{23} \in S_1$ (By stability of S_1). Here, another two cases need to be investigated whether the label $l_{23} \in S_2$ or $\notin S_2$:

   (a) $l_{23} \notin S_2$ is impossible since $l_{23} \in S_1$ (By $S_1 \subseteq S_2$).

   (b) Suppose $l_{23} \in L_2$: by unfolding the synchronization rule we would obtain the three transitions $q_i \xrightarrow{l_i} q'_i$ where $l_{23} = l_2 \bowtie l_3$ (by $\bowtie$ commutativity) and $i \in \{1, 2, 3\}$. Now by applying twice the synchronization rule (since $l_1 \in S_1$ and $l_2, l_3 \in S_1 \subseteq S_2$) starting with the composition of $q_1 \xrightarrow{l_1} q'_1$ and $q_2 \xrightarrow{l_2} q'_2$ and then composing their result with $q_3 \xrightarrow{l_3} q'_3$ we would reach $((q_1, q_2), q_3) \xrightarrow{(l_{12}) \bowtie l_3} ((q'_1, q'_2), q'_3)$. Since LS is associative then $(l_{12}) \bowtie l_3 = l$.

2. Suppose $l \notin S_1$: since $alphabet(\mathcal{L}_1) \subseteq S_1$ only the $\texttt{interleaving}_R$ rule applies leading to the transition $(q_2, q_3) \xrightarrow{l} (q'_2, q'3)$. Here two cases need to be investigated whether $l \in S_2$ or $\notin S_2$:

   (a) Suppose $l \in L_2$: by unfolding the synchronization rule we would obtain the transitions $q_i \xrightarrow{l_i} q'_i$ where $l = l_2 \bowtie l_3$ (by $\bowtie$ commutativity) and $i \in \{2, 3\}$. Knowing that $alphabet(\mathcal{L}_1) \subseteq S_1$ and $l_2, l_3 \in S_2$ (By stability of S_2) then by applying the $\texttt{interleaving}_R$ on the composition of q_1 and $q_2 \xrightarrow{l_2} q'_2$ we would reach $(q_1, q_2) \xrightarrow{l_2} (q_1, q'_2)$ and then composing it with $q_3 \xrightarrow{l_3} q'_3$ we would reach $((q_1, q_2), q_3) \xrightarrow{l_2 \bowtie l_3} ((q_1, q'_2), q'_3)$ where $l_2 \bowtie l_3 = l$ (by $\bowtie$ commutativity).

   (b) Suppose $l \notin S_2$: knowing that $alphabet(\mathcal{L}_2) \subseteq S_2$, only the $\texttt{interleaving}_R$ rule applies leading to the transition $q_3 \xrightarrow{l} q'_3$. Now applying twice the $\texttt{interleaving}_R$ rule we would reach $((q_1, q_2), q_3) \xrightarrow{l} ((q_1, q_2), q'_3)$.

By following a similar reasoning, we prove $(\mathcal{L}_1 \langle \bowtie \rangle \mathcal{L}_2) \langle \bowtie \rangle \mathcal{L}_3 \simeq_{R^{-1}} \mathcal{L}_1 \langle \bowtie \rangle (\mathcal{L}_2 \langle \bowtie \rangle \mathcal{L}_3)$. □

LTS Sum

Given $LS_1 = \langle L_1, L_{E_1}, \bowtie_1 \rangle$ and $LS_2 = \langle L_2, L_{E_2}, \bowtie_2 \rangle$, and two LTS $\mathcal{L}_{1_{LS_1}} = \langle Q, Q^0, T_1, \alpha_1, \beta_1, \lambda_1 \rangle$ and $\mathcal{L}_{2_{LS_2}} = \langle Q, Q^0, T_2, \alpha_2, \beta_2, \lambda_2 \rangle$ defined over the same state space Q, their sum is also an LTS $\mathcal{L}_{LS_1 \oplus LS_2}$ defined over the same state space but in which the transitions are the sum of the transitions of $\mathcal{L}_{1_{LS_1}}$ and $\mathcal{L}_{2_{LS_2}}$. Formally the LTS sum [38] is defined as follows :

$$\langle Q, Q^0, T_1, \alpha_1, \beta_1, \lambda_1 \rangle \bigcup \langle Q, Q^0, T_2, \alpha_2, \beta_2, \lambda_2 \rangle = \langle Q, Q^0, T_1 \uplus T_2, \alpha, \beta, \lambda \rangle$$

$$\text{where} \quad \begin{array}{lll} \alpha(t_1^\bullet) \mapsto \alpha_1(t_1) & \beta(t_1^\bullet) \mapsto \beta_1(t_1) & \lambda(t_1^\bullet) \mapsto \lambda_1(t_1)^\bullet \\ \alpha({}^\bullet t_2) \mapsto \alpha_2(t_2) & \beta({}^\bullet t_2) \mapsto \beta_2(t_2) & \lambda({}^\bullet t_2) \mapsto {}^\bullet\lambda_2(t_2) \end{array}$$

LTS Label Structure Transformations.

The label structure of an LTS may be changed for instance making local a global event (CSP hide) or changing its name (CSP Rename). Here, we consider some LTS labels transformations.

Left(Right) Embedding. For $i, j \in \{1, 2\}$ and $i \neq j$, given $\mathcal{L}_{LS_i} = \langle Q, Q^0, T, \alpha : T \to Q, \beta : T \to Q, \lambda : T \to L_i \rangle$ and LS_j, a left (right) embedding of $\mathcal{L}_{LS_i}$ is a transformation $\uparrow_{LS_i}^{LS_1 \oplus LS_2}$: $\mathcal{L}_{LS_i}, LS_j \to \mathcal{L}_{LS_1 \oplus LS_2}$ where $\mathcal{L} \uparrow_{LS_i}^{LS_1 \oplus LS_2} = \langle Q, Q^0, T', \alpha : T' \to Q, \lambda : T' \to Q, \lambda : T' \to L_1 \uplus L_2 \rangle$ such as T' is defined:

$$i = 1 \quad \frac{t : q \xrightarrow{l_1}_T q', l_1 \in L_1}{t \uparrow^{\oplus} : q \xrightarrow{l_1^\bullet}_{T'} q'} \textsc{Rename}_L \qquad i = 2 \quad \frac{t : q \xrightarrow{l_2}_T q', l_2 \in L_2}{t^{\oplus} \uparrow : q \xrightarrow{{}^\bullet l_2}_{T'} q'} \textsc{Rename}_R$$

Left(Right) Extension. For $i, j \in \{1, 2\}$ and $i \neq j$, given $\mathcal{L}_{LS_i} = \langle Q, Q^0, T, \alpha : T \to Q, \beta : T \to Q, \lambda : T \to L_i \rangle$ and LS_j, a left (right) extension of $\mathcal{L}_{LS_i}$ is a transformation $\uparrow_{LS_i}^{LS_1 \otimes LS_2}$: $\mathcal{L}_{LS_i}, LS_j \to \mathcal{L}_{LS_1 \otimes LS_2}$ where $\mathcal{L} \uparrow_{LS_i}^{LS_1 \otimes LS_2} = \langle Q, Q^0, T', \alpha : T' \to Q, \lambda : T' \to Q, \lambda : T' \to L_1 \times L_2 \rangle$ and T' is defined:

$$i = 1 \quad \frac{t : q \xrightarrow{l_1}_T q', l_1 \in L_1}{t_{l_2}{}^{\otimes} \uparrow : q \xrightarrow{(l_1, \epsilon')}_{T'} q'} \textsc{Extension}_L \qquad i = 2 \quad \frac{t : q \xrightarrow{l_2}_T q', l_2 \in L_2}{t_{l_1} \uparrow^{\otimes} : q \xrightarrow{(\epsilon, l_2)}_{T'} q} \textsc{Extension}_R$$

Left(Right) Projection. For $i \in \{1, 2\}$, given $\mathcal{L}_{LS_1 \oplus LS_2} = \langle Q, Q^0, T, \alpha : T \to Q, \lambda : T \to Q, \lambda : T \to L_1 \uplus L_2 \rangle$, a left (right) projection of $\mathcal{L}_{LS_1 \oplus LS_2}$ is a transformation $\uparrow_{LS_1 \oplus LS_2}^{LS_i}$: $\mathcal{L}_{LS_1 \oplus LS_2} \to \mathcal{L}_{LS_i}$ where $\mathcal{L} \uparrow_{LS_1 \oplus LS_2}^{LS_i} = \langle Q, Q^0, T', \alpha : T' \to Q, \lambda : T' \to Q, \lambda : T' \to L_i \rangle$ such as T':

$$i = 1 \quad \frac{t : q \xrightarrow{l_1^\bullet}_T q', l_1 \in L_1}{t \downarrow_{\oplus} : q \xrightarrow{l_1}_{T'} q'} \textsc{Projection}_L \qquad i = 2 \quad \frac{t : q \xrightarrow{{}^\bullet l_2}_T q', l_2 \in L_2}{t_{\oplus} \downarrow : q \xrightarrow{l_2}_{T'} q} \textsc{Projection}_R$$

Left Embedding Application. Given LS and $\uparrow_{LS_1}^{LS_2}$ a transformation from LS_1 to LS_2, a left embedding application of $\uparrow_{LS_1}^{LS_2}$ is a transformation of a transformation $\Uparrow_{\oplus LS}^{\oplus LS} : \uparrow_{LS_1}^{LS_2} \rightarrow \uparrow_{LS_1 \oplus LS}^{LS_2 \oplus LS}$ that is applied to an LTS $\mathcal{L}_{LS_1 \oplus LS}$ as:

$$\mathcal{L}_{LS_1 \oplus LS}(\uparrow_{LS_1}^{LS_2} \Uparrow_{\oplus LS}^{\oplus LS}) = (\mathcal{L} \uparrow_{LS_1 \oplus LS}^{LS_1}) \uparrow_{LS_1}^{LS_2} \bigcup \mathcal{L} \uparrow_{LS_1 \oplus LS}^{LS}$$

1.2.1 Timed Transition Systems TTS

Definition 1.4 (TTS) *Given a label structure $LS = \langle L, L_E, \bowtie \rangle$, a Timed Transition System TTS over LS is an LTS over $LS \oplus TS$.*

Definition 1.5 (TTS composition) *Thanks to the introduction of our label structure, the TTS composition is the composition of the underlying LTSs.*

1.2.2 Timed Automata TA

We consider a timed automata [15] in which no invariants are associated to its locations (this is close to a timed graph [13] since neither invariants nor committed states are modeled). The transitions are in the form of *guard/event/reset* where the guards contain a conjunction of constraints represented as clock intervals and the reset actions consist in a set of clocks to be reset. This is represented as a product of two label structures. The first manages the synchronization events while the second manages the clock access. Formally, given a label structure $LS = \langle L, \bowtie \rangle$, a set of clock variables C, a timed automata (TA) over LS is an LTS over $LS \otimes CM$ where CM is a clock manager structure. For $I \in \mathbb{I}$ where $\mathbb{I}$ is the set of intervals of $\mathbb{R}^+$, CM is defined as :

$$CM = \left| \bowtie \right| \begin{array}{l} \langle L = \{R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}!, R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}? \mid c, c', c'' \subseteq C \wedge c' \cap c'' = \emptyset\}, \\ \quad R_{\vec{c_1}}^{\vec{I_1}} \blacktriangleright Z_{\vec{c_1}'} U_{\vec{c_1}''}! \bowtie R_{\vec{c_2}}^{\vec{I_2}} \blacktriangleright Z_{\vec{c_2}'} U_{\vec{c_1}''}! \mapsto \\ \quad R_{\vec{c_1}.\vec{c_2}}^{\vec{I_1}.\vec{I_2}} \blacktriangleright Z_{\vec{c_1}' \cup \vec{c_2}'} U_{\vec{c_1}'' \cup \vec{c_2}''} \textbf{ if } c_1' \cap c_2'' = \emptyset \wedge c_1'' \cap c_2' = \emptyset \\ \quad R_{\vec{c_1}}^{\vec{I_1}} \blacktriangleright Z_{\vec{c_1}'} U_{\vec{c_1}''}! \bowtie R_{\vec{c_2}}^{\vec{I_2}} \blacktriangleright Z_{\vec{c_2}'} U_{\vec{c_1}''}? \mapsto \\ \quad R_{\vec{c_2}}^{\vec{I_2}} \blacktriangleright Z_{\vec{c_2}'} U_{\vec{c_1}''}? \textbf{ if } \vec{c_1} \subseteq \vec{c_2} \wedge (\forall c \in \vec{c_1}, I_2(c) \subseteq I_1(c)) \wedge c_1' \subseteq c_2' \wedge c_1'' \subseteq c_2'' \\ \quad R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}? \bowtie R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}? \mapsto R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}? \rangle \end{array}$$

The CM events are used to model the communication between a timed automata and a process managing the clocks (See Semantics of TA). The clock access orders are sent (!) by the timed automata and received (?) by the clock manager. An event $R_{\vec{c}}^{\vec{I}} \blacktriangleright Z_{\vec{c}'} U_{\vec{c}''}$ means that if the clocks of $\vec{c}$ are in the corresponding intervals of $\vec{I}$ then the clocks of $\vec{c'}$ are reset while the clocks of $\vec{c}''$ are kept unchanged. The composition of two sending events leads to a third one in which its vectors are respectively the concatenation (resp. union when the order does not matter) of the composed events vectors. A receiving event is exclusively used to model the clock manager events. The composition of a sending and a receiving event is defined when the sending sets are included in the receiving ones. Finally, two receiving events are only composed when they are identical.

Definition 1.6 (TA Composition) *Thanks to our label structure, the TA composition is defined as the composition of the underlying LTS systems.*

TA Semantics.

The semantics of a timed automaton *ta* is given via a composition with a clock manager *Clk* defined over $CM \oplus TS$ (Fig 1.1). *Clk* has two types of transitions. Transitions of time evolution in which after each possible delay all of the clocks are incremented by the amount of this delay. Transitions in which certain clocks are checked against their timing intervals before resetting others. Since *Clk* is diagonal, idempotent and deterministic, it follows that $Clk\langle \bowtie \rangle Clk \simeq Clk$.

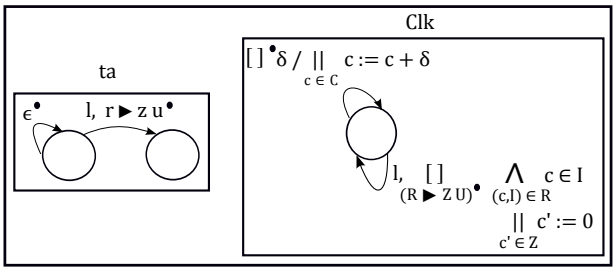


Figure 1.1 – *Semantics of TA via a Composition of Two LTS*

Furthermore, since the LTS composition is defined over the same label structure, then the label structures on which *ta* and *Clk* are defined have to be adapted. The systems *ta* and *Clk* are then extended respectively with ϵ transitions so that they both become defined over $LS \otimes CM \oplus TS$. More precisely, by using the left embedding transformation function with the TS label structure over *ta* we would obtain $ta_{LS\otimes CM\oplus TS}$ and by applying the left embedding application over *Clk* along with the transformation $\uparrow_{CM}^{LS\otimes CM}$ we would obtain $Clk_{L\otimes CM\oplus TS}$. This is formally defined as $[\![ta]\!] = (ta \uparrow_{LS\otimes CM}^{LS\otimes CM\oplus TS} \langle \bowtie_{(LS\otimes CM)^\bullet} \rangle Clk \uparrow_{CM\oplus TS}^{LS\otimes CM\oplus TS})$ where the composition of *ta* and *Clk* is synchronous over $LS \otimes CM$ while *Clk* is asynchronous over TS.

Chapter Two

Appendix B : BPEL To FIACRE Patterns

2.1 Modeling the WSDL

The interaction with the environment is supported by Partnerlinks. In BPEL, each Partnerlink may contain two roles (`myRole` and `partnerRole`) typed with `Porttype`. Each of the `Porttype` declares several operations used to receive (Input) or send (Output) messages (Fig 2.1).

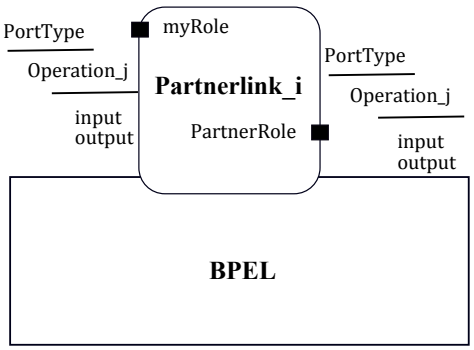


Figure 2.1 – *Connections of BPEL*

Consequently, this structure is modeled in FIACRE by creating two different enumerated types named `inputs` and `outputs` used to model respectively the inputs and the outputs of each operation. The type `inputs` (resp. `outputs`) will be the union of the type of :

- the `input` (resp. `output`) arguments of operations of the `myRole` of every Partnerlink (1).
- the `output` (resp. `input`) arguments of operations of the `partnerRole` of every PartnerLink (2).

More precisely, the types encoded in FIACRE in order to model the WSDL part are as follows :

1. Type `inputs` : The name of the constructor is built from the name of the partnerlink and a suffix `recv_call` (vs. `recv_result`) in

case (1) (resp. in case (2)). As for the name of the constructor's type argument, it is built using the name of the partnerlink with the suffix `type_args` (vs. `type_results`) in case (1) (resp. in case (2)). The FIACRE code of the `inputs` type is shown in the following :

```
type inputs_BPELName is union
     partnerlinkName_recv_call of partnerlinkName_type_args
   | partnerlinkName_recv_result of partnerlinkName_type_results
   | ...
end
```

2. Type `outputs` : The name of the constructor is built from the name of the partnerlink and a suffix `send_call` (vs. `send_result`) in case (2) (resp. in case (1)). As for the name of the constructor's type argument, it is built using the name of the partnerlink with the suffix `type_args` (vs. `type_results`) in case (2) (resp. in case (1)). The FIACRE code of the `outputs` type is shown in the following :

```
type  outputs_BPELName is union
      partnerlinkName_send_call of partnerlinkName_type_args
    | partnerlinkName_send_result of partnerlinkName_type_results
    | ...
end
```

3. Type `partnerlink_type_args` (vs. `partnerlink_type_results`) : Each of the partnerlinks types are also declared as a union type containing all of the porttypes of the corresponding partnerlink. The name of the constructor is built from the name of the porttype and a suffix `partnerlink_args` (resp. `partnerlink_result`). As for the name of the constructor's type argument, it is built using the name of the porttype with the suffix `partnerlink_type_args` (resp. `partnerlink_type_result`). The corresponding FIACRE code is shown in the following :

```
type partnerlinkName_type_results is union
        PortTypeName_partnerlinkName_result of
                PortTypeName_partnerlinkNam_type_result
...
end

type partnerlinkName_type_args is union
        PortTypeName_partnerlinkName_args of
                PortTypeName_partnerlinkNam_type_args
...
end
```

4. Type `porttype_partnerlink_type_args` : Each of the porttype types are also declared as a union type containing all of the operations of the corresponding partnerlink. Since an operation in BPEL always has an input attribute (whether the operation is synchronous or asynchronous), then the name of the constructor is built from the name of the operation and a suffix

porttype_partnerlink_args. The type of the constructor will correspond to a record containing an abstracted version of the actual BPEL message types. The fields of each record will correspond to the parts of the modeled BPEL message type. The corresponding FIACRE code is shown in the following :

```
type PortTypeName_partnerlinkName_type_args is union
operationName_PortTypeName_partnerlinkName_type_args of
  record partName1: abstract_type end,
  ...
end
```

5. Type porttype_partnerlink_type_results : Each of the porttype types are also declared as a union type containing all of the operations of the corresponding partnerlink. However, since an output attribute for BPEL operations is optional (required only for synchronous operations), then two cases are possible :

   (a) In the case no output attribute is present, then the porttype_partnerlink_type_results will be typed with a constructor indicating that this fact. The name of the constructor is built of a concatenation of the literal unused and a suffix porttype_partnerlink_result.

   (b) In the other case, the name of the constructor is built from the name of the operation and a suffix porttype_partnerlink_result. As for the name of the constructor's type argument, it is built of a concatenation of the literal op_result with the suffix containing the name of the porttype and the name of the operation (porttype_operation). The corresponding FIACRE code is shown in the following :

```
type PortTypeName_partnerlinkName_type_result is union
operationName_PortTypeName_partnerlinkName_type_result of
  op_result_PortTypeName_operationName,
  ...
end
```

6. Type op_result_porttype_operation : This type consists of the possible results of a synchronous operation. This is why, it will be the union of two kinds of constructors :

   (a) The actual result of the message. The name of the constructor is built from a concatenation of the literal resultat and the suffix porttype_operation. This constructor will be typed with a record containing an abstracted version of the actual BPEL message types. The fields of each record will correspond to the parts of the modeled BPEL message type.

(b) The BPEL faults. A fault in BPEL has a fault name, and a variable to give additional information to the corresponding fault. In Fiacre, a fault will be typed by the name of the fault and by a constant that will represent the type of the variable associated to the fault. We then filter the faults based on their names, and the type of the variable associated with it (in case it exists). The name of the constructor is built from a concatenation of the literal `error` and the suffix `porttype_operation`. This constructor will be typed with the `fault` type which will be the union of all the user fault names in a BPEL process and some of the BPEL default fault names. These fault names will be typed with *anytype* that is declared as Boolean in order to indicate the existence of a specified fault. The corresponding FIACRE code is shown in the following :

```
type anytype is bool

type fault is union user_fault of bp_anytype | bp_F_variableError
| bp_F_expressionError end

type op_result_PortTypeName_operationName is union
        resultat_PortTypeName_operationName
                of record partName1: abstract_type end,
...
| error_PortTypeName_operationName of fault
...
end
```

Finally like we have already said, having discarded data values, a constant will be created for each data type that represents the type of the message. So the operations will be typed by this constant that represent the BPEL type of the message. Note that for abstraction purposes, only Boolean and enumerated types are preserved during this transformation. Actually, with respect to model checking purposes, considering the actual values is not realistic because of state explosion.

2.1.1 Example of WSDL Modeling

We give an example of a partnerlink containing one portType used in a synchronous communication. Therefore the operation has two types of messages, an Input and an Output message. Based on what was explained earlier in this section, 2 types will be created in order to model this partnerlink in Fiacre. The first type, *inputs* will model the input message (POMessage) of the partnerlink, as for the type *outputs* it will model the output message (InvMessage or a fault CannotCompleteOrder).

```
<wsdl:message name="POMessage">
        <wsdl:part name="customerInfo"
                type="sns:customerInfoType"/>
        <wsdl:part name="purchaseOrder"
                type="sns:purchaseOrderType" />
</wsdl:message>
<wsdl:message name="InvMessage">
        <wsdl:part name="IVC"
                type="sns:InvoiceType" />
</wsdl:message>
<wsdl:message name="orderFaultType">
        <wsdl:part name="problemInfo"
                element="sns:OrderFault" />
</wsdl:message>

<wsdl:portType name="purchaseOrderPT">
        <wsdl:operation name="sendPurchaseOrder">
                <wsdl:input message="pos:POMessage" />
                <wsdl:output message="pos:InvMessage" />
                <wsdl:fault name="cannotCompleteOrder"
                                message="pos:orderFaultType" />
        </wsdl:operation>
</wsdl:portType>

<plnk:partnerLinkType name="purchasingLT">
        <plnk:role name="purchaseService"
                        portType="pos:purchaseOrderPT" />
</plnk:partnerLinkType>
```

```
type purchaseOrderPT_purchasing_type_args is union
sendPurchaseOrder_purchaseOrderPT_purchasing_args of
  record purchaseOrder:record purchaseOrderType:unit end,
  customerInfo:record customerInfo:abs_string end end end

type op_result_purchaseOrderPT_sendPurchaseOrder is union
resultat_purchaseOrderPT_sendPurchaseOrder of
  record IVC:record InvoiceType:unit end end
| error_purchaseOrderPT_sendPurchaseOrder of fault end

type purchaseOrderPT_purchasing_type_result is union
sendPurchaseOrder_purchaseOrderPT_purchasing_result of
p_op_result_purchaseOrderPT_sendPurchaseOrder end

type purchasing_type_results is union
purchaseOrderPT_purchasing_result of
purchaseOrderPT_purchasing_type_result end

type purchasing_type_args is union
purchaseOrderPT_purchasing_args of
purchaseOrderPT_purchasing_type_args end


type outputs_purchaseOrderProcess is union
        invoicing_send_call of invoicing_type_args
        | shipping_send_call of shipping_type_args
        | scheduling_send_call of scheduling_type_args
        | purchasing_send_result of purchasing_type_results
        | invoicing_send_result of invoicing_type_results
        | shipping_send_result of shipping_type_results end

type inputs_purchaseOrderProcess is union
        purchasing_recv_call of purchasing_type_args
        | invoicing_recv_call of invoicing_type_args
        | shipping_recv_call of shipping_type_args
        | invoicing_recv_result of invoicing_type_results
        | shipping_recv_result of shipping_type_results
        | scheduling_recv_result of scheduling_type_results end
```

2.2 Basic Activities

2.2.1 Rethrow

We can also transmit errors explicitly to the next enclosing scope using the *rethrow* activity. It can be used only within a fault handler (*catch* and *catchAll*) to rethrow the caught fault. For this fact, the signature of this activity is slightly different than other activities. The difference results from the fact that this activity needs to receive the fault at first before it is capable to rethrow it. That is why we create two ports of type fault in the *rethrow* activity. The first f is used to receive the fault and the second $f2$ to rethrow it.
The Fiacre process pattern for *rethrow* is given in Figure 2.2.

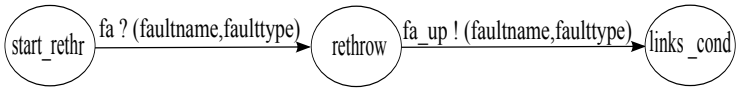


Figure 2.2 – *Rethrow pattern*

2.2.2 Exit

The *exit* activity terminates immediately the business process by disabling all of its activities without any fault handler being invoked. In Fiacre, this is done by setting all the shared variables *error* of all the scopes to *exit* and by setting all the *stop* variables of the scopes to *true*, starting from the *root_stop* variable. In order to do this, the pattern of the exit activity consist of a synchronization on an event `exit` directly with the fault handler of the root process.

The Fiacre process pattern for this activity is given in Figure 2.3.

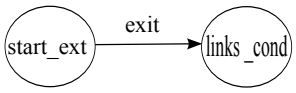


Figure 2.3 – *Exit pattern*

2.2.3 Compensate

The compensate activity is used in order to execute all of the compensation handlers of the child scopes. It may used inside the FCT handlers of BPEL. In Fiacre this is done by valuating the *comp* variable of all the child scopes to True (Fig. 2.4). The compensate activity will then do a blocking wait until all of the the *comp* variables are set back to False by the corresponding compensation handlers. We note as well that a compensate activity cannot throw any faults. Even though that the executed compensation handlers may do. We have decided that these thrown faults are sent directly to the Fault handler which will treat them or propagate them.

2.2.4 Compensate Scope

The compensate scope is similar to the compensate activity. It is used to compensate a specific scope by specifying its name. In Fiacre we know ex-

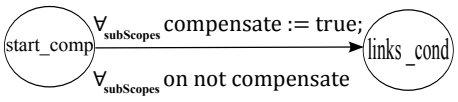


Figure 2.4 – *Compensate pattern*

actly what *comp* variable we set true by inspecting the BPEL code (Fig. 2.5).

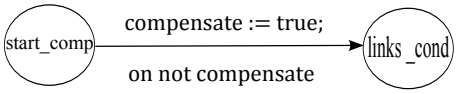


Figure 2.5 – *Compensate Scope activity*

2.3 Structured Activities

2.3.1 While and RepeatUntil

The `while` and the `repeatUntil` activities allow for repeated execution of a nested activity. While the *while*'s nested activity may not be executed in case the condition initially evaluates to *false*, the *repeatUntil*'s nested activity is executed for at least one time. In Fig. 2.6 we show the pattern associated to each of these activities.

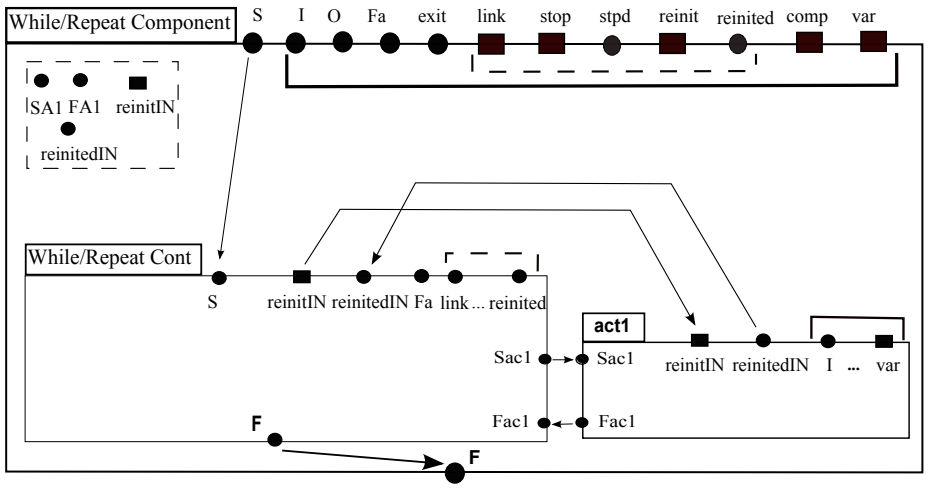


Figure 2.6 – *While/RepeatUntil component*

Depending on whether the activity is a *while* or a *repeatUntil*, a different controller is created (Fig. 2.7). Here we note that conversely to the controllers already seen, we show the `init` state which is normally a state of the common behavior. This is done because the `init` state of the while and the repeat controllers have a special treatment related to the reinitialization of activities.

- For the case of the while controller: once at the `init_while` state, if a reinitialization demand is received, the while controller will signal

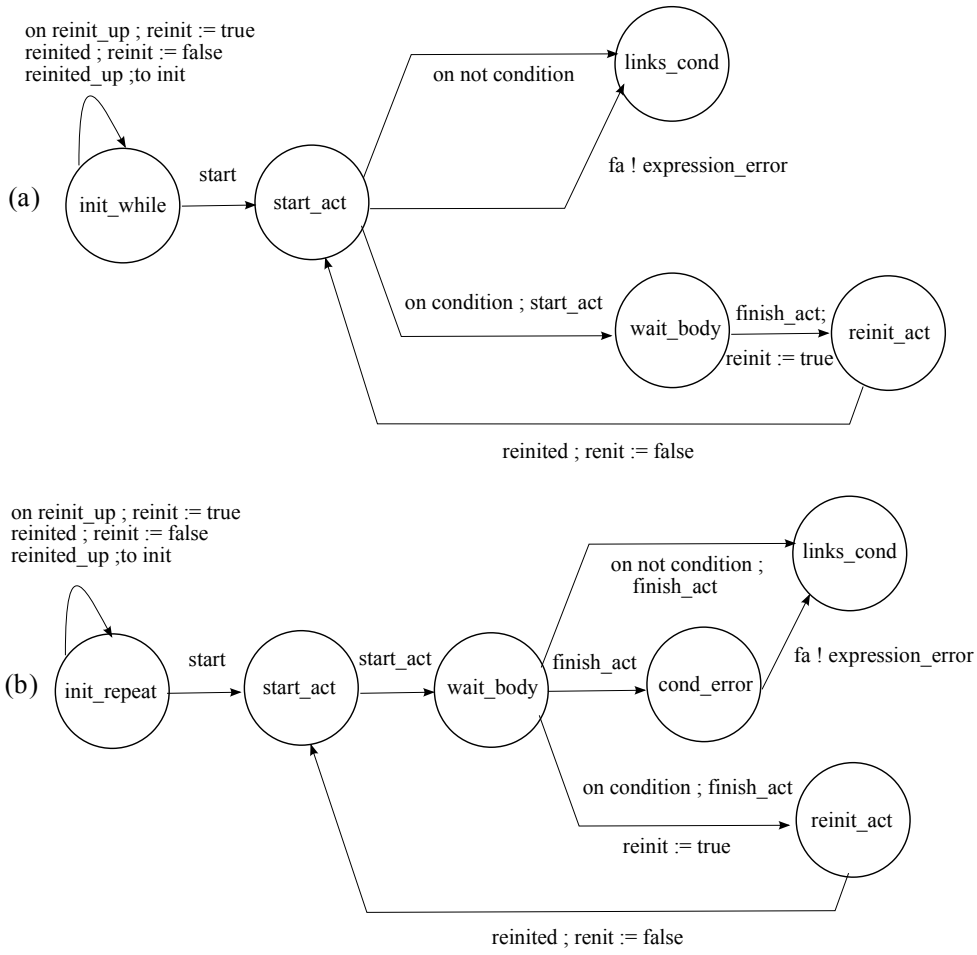


Figure 2.7 – *(a) While controller / (b) Repeat Until controller*

this fact to its nested activities and will wait until the reinitialization is done before transmitting a response to its upper activity.

- At the *start_act* state, three transitions are possible based on the condition of the `while` activity. [1] :

  1. in case the condition is violated, the process can transit directly to the *links cond* state in order to set the outgoing links of the while activity.
  2. in case the transition is satisfied, the process can transmit the start signal of the nested activity attached to the while before transiting to the *wait_body* state.
  3. The evaluation of the condition can fail and in this case a standard BPEL fault will be throw (validExpressionValue).

[1] Here we note that the condition of Fig. 2.7 may be modeled in FIACRE in case it manipulates simple data types such as Boolean. However, in case the data type were complex such as strings, since we abstract these data the choice between the satisfaction of the condition or not would be non-deterministic (as if we replace the condition by `true`)

This corresponds to the transition labeled with the fault `expression_error`

Once the while controller is at the *wait_body* state, it will do a blocking wait until the finish signal of the nested activity is received, then it will transit to the *reinit_act* state in order to reinitialize the activities nested in the while activity so that multiple instances of the while body may be executed[2]. The reinitialization is done by valuating the variable reinitialize *reinit* to true and then it awaits that the activities synchronize on the port *reinited*.

- For the case of the repeatUntil controller, a slight difference is made in order to force the execution of the nested activity for at least one time before exiting the loop. More precisely, at the *start_act* state of the *repeat Until* controller (Figure 2.7 (b)) the body activity is executed directly before the test of the condition is made.

2.3.2 If

The *If* activity of BPEL controls the conditional execution of the nested activities. The *If* activity contains multiple (if and else if) conditions to which a nested activity is associated. The conditions are checked sequentially. Depending on which of the conditions evaluates to True its associated activity is executed. For the last branch of the *IF* activity, it corresponds to the "else" branch. The activity associated to this branch is executed in case all of the preceding conditions evaluated to False.

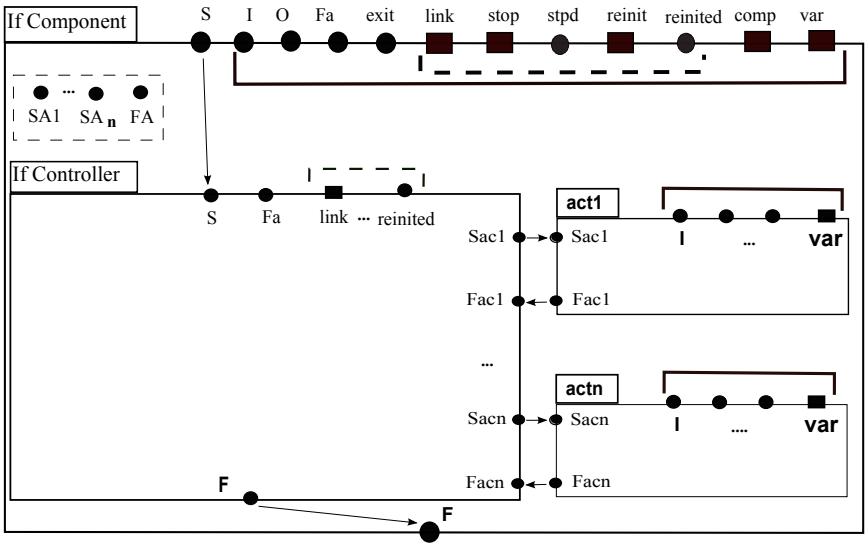


Figure 2.8 – *If component*

Whenever the start signal of the *If* is received, the *If* controller will proceed by sending the start signal to one of the nested activities (one of SA1 to SA_n in Figure 2.8). It will then do a blocking wait until one of the

[2]All the variables used by the Fiacre components are reset to their initial values.

finish signal of the executed activity is received (FA). [3]. Once it is received, the *If* controller will signal the finish of the *If* activity.

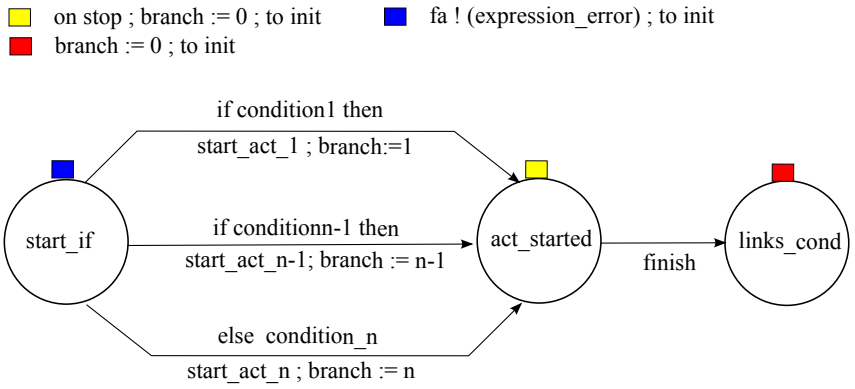


Figure 2.9 – *If controller*

In Fig 2.9 we show the controller of the *If* activity . The BPEL `If` activity is modeled in FIACRE as an `if` statement. In each branch of this statement the BPEL conditions are embedded. In case the condition `condition1` is satisfied, the first transition is taken. Otherwise the second condition is tested and so on. The *If* controller starts by reading the condition of the first branch. The evaluation of the condition can fail and in this case a standard BPEL fault will be thrown (`validExpressionValue`). [4] Each transition of the `start_if` state :

1. The process can transmit the start signal (`start_act_1 to start_act_n`) of the corresponding nested activity attached to the satisfied branch of the `If` activity before transiting to the `act_started` state. Once the `If` controller is at the `act_stated` state, it will wait for the finish signal of the nested activity to be received, then it updates the value of a local variable `branch` with an integer that points to the chosen branch before transiting to the `links_cond` state. This `branch` variable will be used to indicate which branch has been chosen by the `If` activity, so that the other activities that are associated to the other branches are informed that they are no longer activated. In this case, positive values of the `skip` variable are transmitted to the unchosen activities.

2. In case the process got to the last transition of the `start_if state` (corresponding to the else branch) -meaning that non of the preceding conditions were satisfied- it signals the start of the activity attached to the else branch before eventually transiting to the `link_cond` state in order to evaluate the outgoing links of the `If` activity.

Finally, whenever the activity is started (state `act_started`) and a stop demand is asked, the `branch` variable needs to be reinitialized to

[3] We note here that all the activities share a single finish signal (FA), since the choice between the execution of one of the nested activities of the `IF` is exclusive

[4] Again, when the BPEL data manipulate complex types, these data are abstracted in FIACRE. That what makes the choice between throwing a fault or transiting to the `branch1` state non-deterministic.

o. This is because in this case none of the nested activities have updated their outgoing links . Also at the `links_cond` state, the `branch` variable is reinitialized in order to be ready for a new execution.

2.3.3 Pick

The *pick* activity waits for a message or an alarm to occur. A nested activity is associated to each of the received events. It is similar to the event handler with the only difference that the `pick` activity waits for exactly one message or for a timeout to occur. Hence, the nested activity associated to the received message is executed for exactly one time. In the pick component (Figure 2.10), the pick controller will signal the start ($SA1 \cdots SA_n$) depending on the received event. It will then waits for the executed activity to finish in order to signal the finish of the pick.

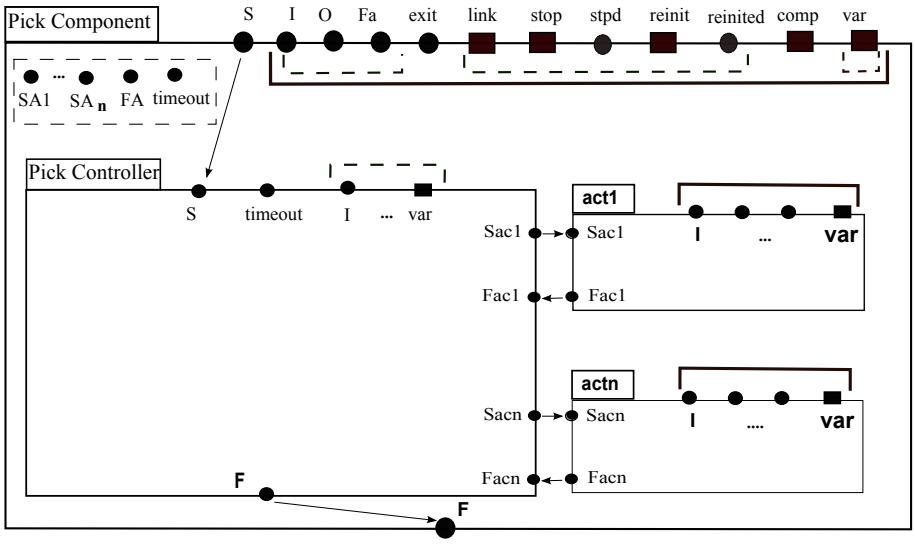


Figure 2.10 – *Pick component*

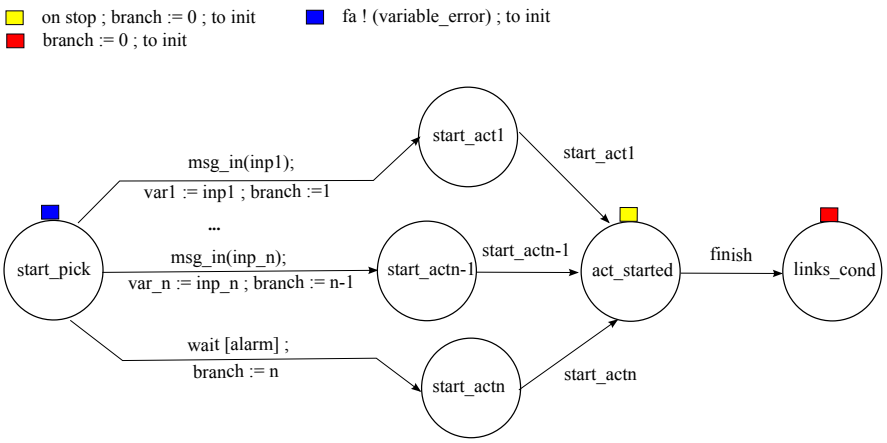


Figure 2.11 – *Pick controller*

The controller (Figure 2.11) of a `Pick` activity contains two types of branches (a message event and an alarm). The `Pick` controller will do

a blocking wait until an event of the two occurs [5]. Once one of these events occur, the pick controller will execute the associated activity and then it will have the same behavior of the If controller concerning the branch local variable. We also note that the alarm event of the pick will be treated exactly the same way as the notion of time is treated in the *wait* activity. Here we note that unlike the If controller, at the `start_pick` state a FIACRE `select` statement is modeled because the choice between the events is non-deterministic. Finally, the explanation concerning the `branch` variable in the `If` controller also holds here.

2.3.4 Full Scope With Termination and Compensation Handlers.

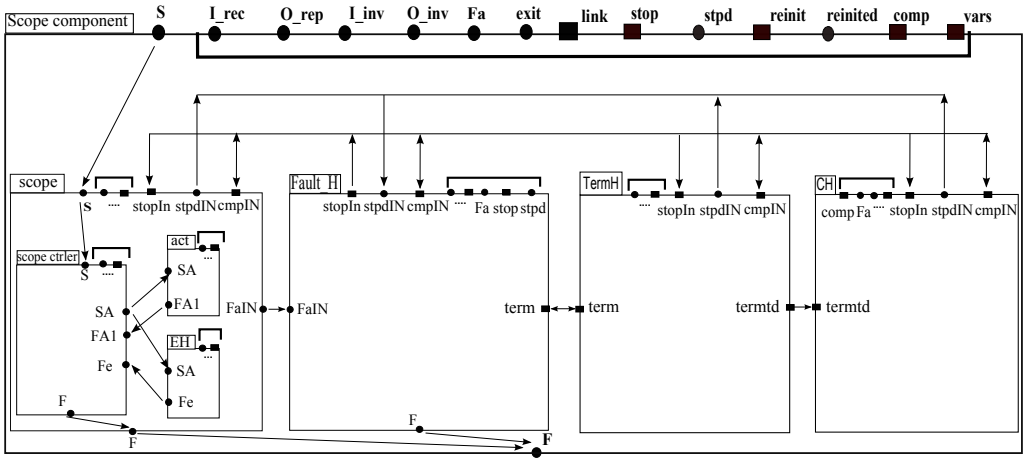


Figure 2.12 – *Scope Component*

Termination Handler: The termination handler is only executed when an enclosed scope is forced to terminate by an enclosing scope. In FIACRE, this is modeled by the variable `term` that is valuated to True by the scope's fault handler whenever a forced termination is demanded. In addition, the scope should be in running mode in order for it to be terminated. In FIACRE, these conditions are checked by the termination controller (embedded in the TH component). If these conditions are fulfilled, then the termination handler is executed. All the faults thrown inside of the termination handler are treated internally and are never propagated. This is why no fault ports appear in the interface of the termination handler. Moreover, all the faults thrown inside of the termination handler are treated internally and never propagated that is why no fault ports appear in the interface of the termination handler. In FIACRE this is done by creating a local fault port *(Fa2)*, a stop variable *(stop2)* and a stop port *(stpd2)*. We note that in our encoding, the innermost-first termination order of BPEL is respected. It is so because the `term` variable is reset to False only

[5]Here we note that the alarm event represents a relative BPEL time. That is the `For duration` attribute is used. It is treated in FIACRE as a wait instruction having the same minimal and maximal bound of the specific BPEL alarm value **`wait`** `[alarm, alarm]`. More details about the modeling of the BPEL notion of time and especially the absolute time is given later in this section (see section 2.5.5

after the execution of the termination handler. Furthermore, it will valuate another variable to True (namely termted). This variable is used by the compensation handler as a certification that the termination handler of the scope has been executed.

Compensation Handler: The compensation handler is triggered by a compensate or compensateScope activity embedded in one of the Fault-Compensation-Termination handlers of the enclosed scope. In FIACRE, this is modeled by valuating the `comp` variable to True. Furthermore, the compensation handler of a scope is only executed if the associated scope has already been completed normally. This condition is verified by the compensation handler controller before it is able to execute its body activity. Unlike the termination handler, the faults thrown by the activities of the compensation handler are propagated to the scope which has called the faulted compensation handler. This is done by synchronizing on the `Fa` port shared with the scope component interface. Furthermore, in BPEL, the compensation of scopes occurs in the reverse order of their completion. In FIACRE, a simple way to handle such compensation order is to handle it non-deterministically. Otherwise, we have to implement a dynamic mechanism as it is done by [47].

2.4 BPEL Handlers

2.4.1 Termination Handler

Termination Handler Component The component of the termination handler in Fiacre is given in Figure 2.13. It consists of two parts, a termination controller and its nested activity.

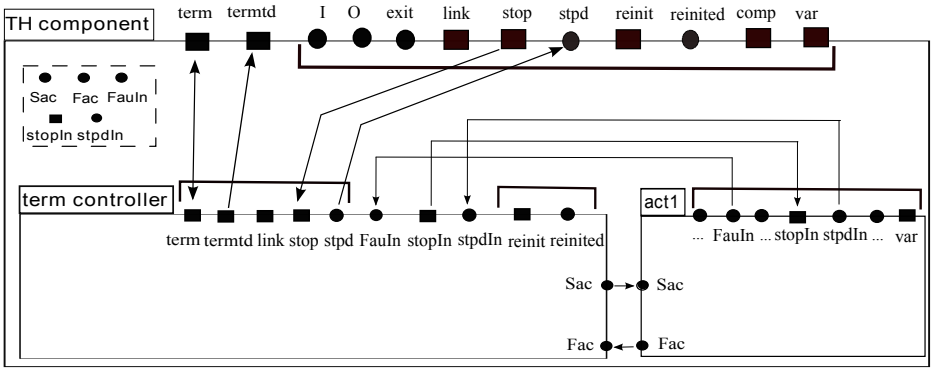


Figure 2.13 – *Termination Handler component*

User Defined Termination Handler Controller A User defined termination controller 2.13may embed any activity.

Default Termination Handler Controller A default termination controller which will only embed a compensate activity.

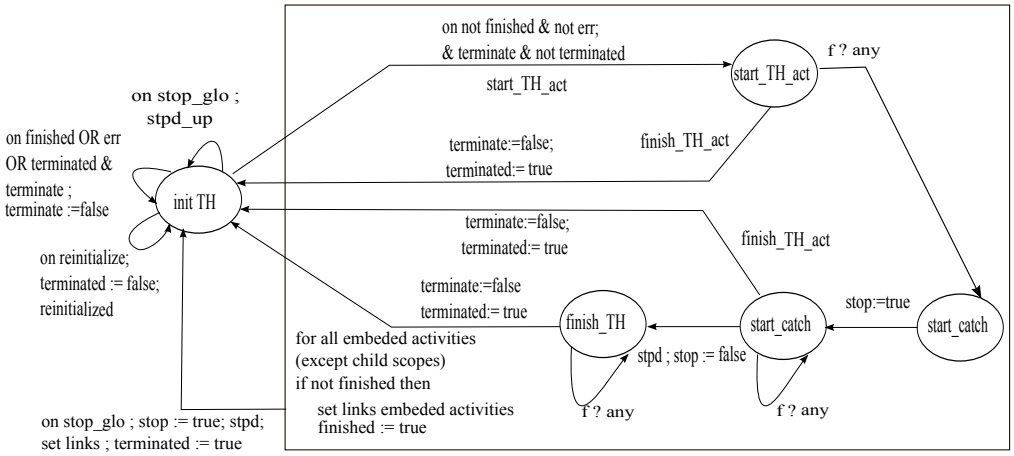


Figure 2.14 – *User defined termination Handler controller*

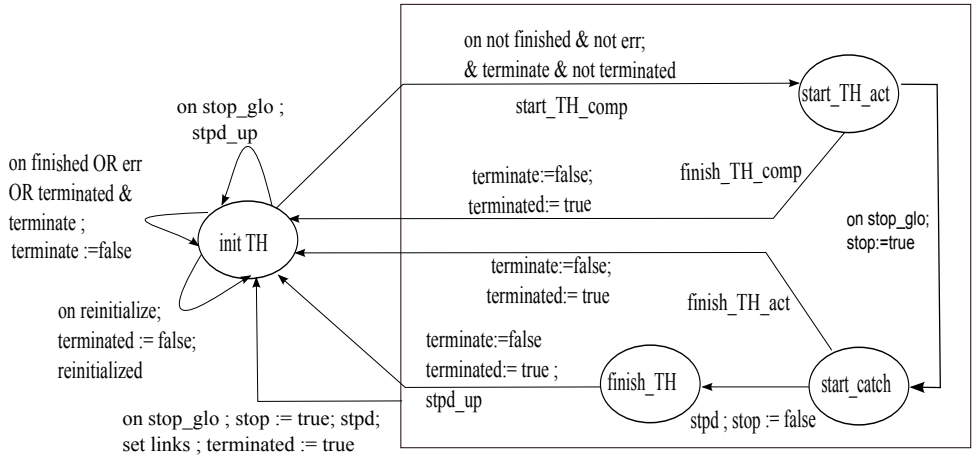


Figure 2.15 – *Default termination handler controller*

2.4.2 Compensation handler

A scope may be asked to be compensated by running its compensation handler. In Fiacre, the component of the compensation handler (Fig 2.16) is similar to other handlers and structured activities. It consists of two parts which are a compensation controller and the nested activities. The start of the compensation handler is treated by its compensation controller (Fig. 2.17 and Fig. 2.18).

User Defined Compensation Controller A user defined compensation handler is a wrapper in which we may nest any type of activities.

Default Compensation Controller In the case that a compensation handler for a scope is not specified explicitly, a default compensation handler is created that only contains a compensate activity.

2.4.3 Full Fault Handler : With Termination and Compensation Handlers

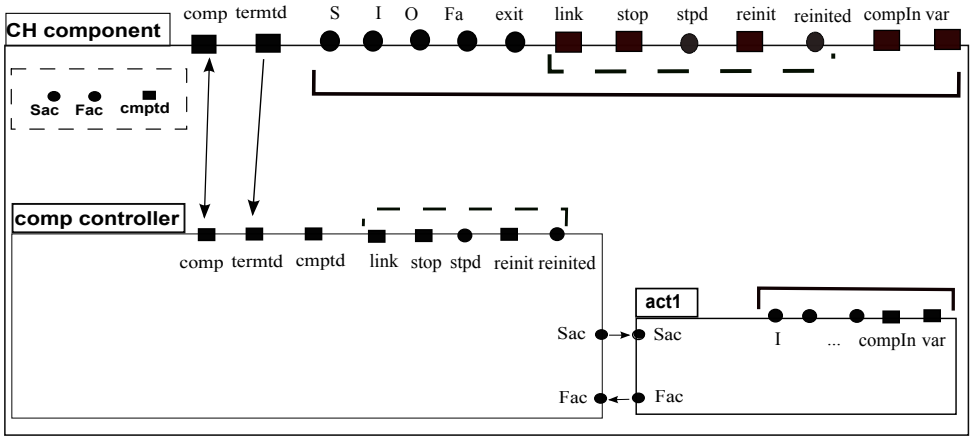


Figure 2.16 – *Compensation handler*

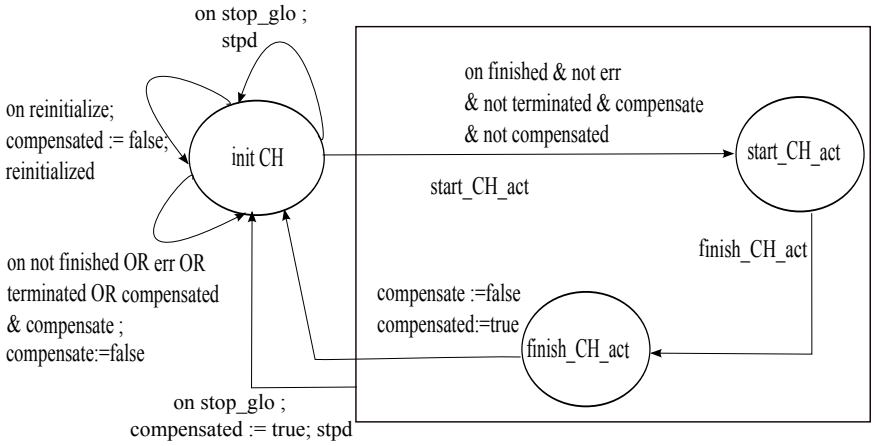


Figure 2.17 – *User defined compensation handler*

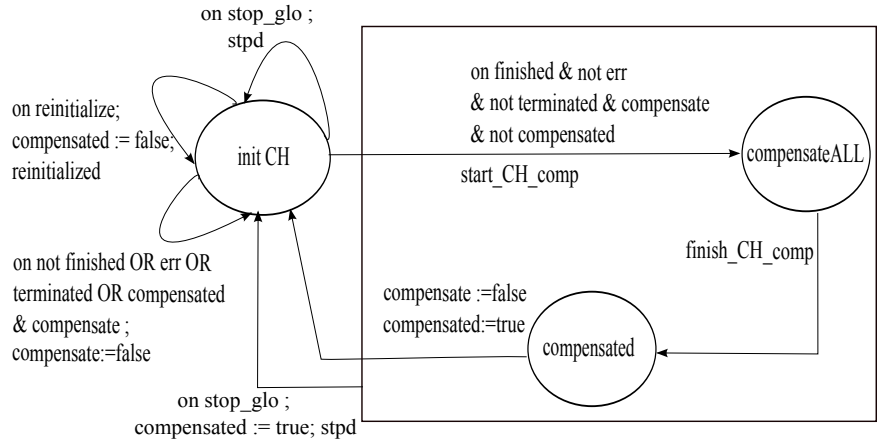


Figure 2.18 – *Default compensation handler*

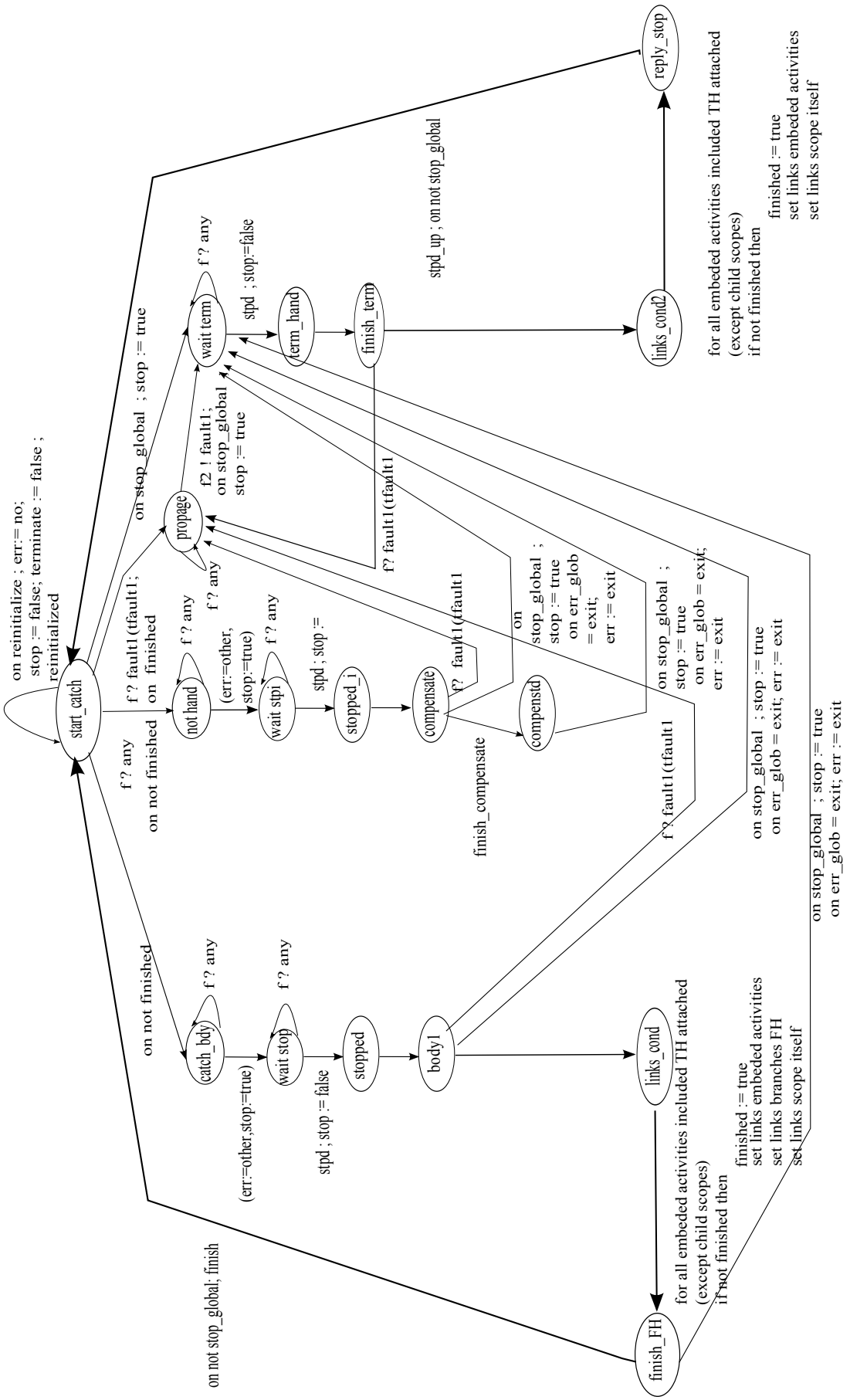


Figure 2.19 – *Fault Handler Controller : No CatchAll*

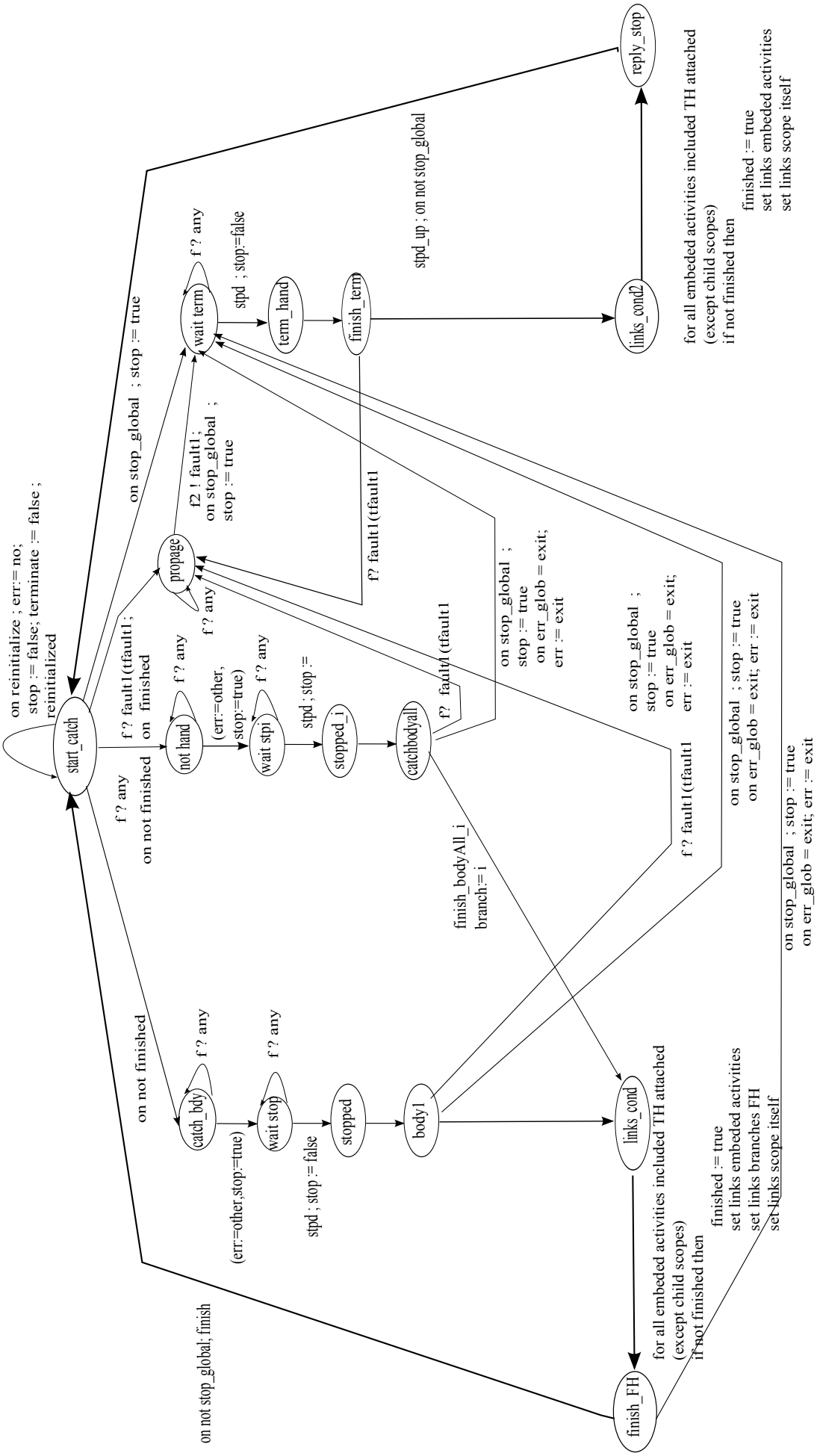


Figure 2.20 – *Fault Handler Controller : CatchAll*

Bibliography

[1] http://ecdar.cs.aau.dk/. (Cité pages 42 et 143.)

[2] http://research.petalslink.org/display/itemis/itemis+overview. (Cité pages x, 6 et 122.)

[3] http://www.w3.org/tr/soap/. (Cité page 84.)

[4] http://www.w3.org/xml/schema. (Cité page 85.)

[5] Review of "concurrent and real-time systems: the csp approach" by steve schneider. wiley 1999. *SIGACT News*, 35:4–12, June 2004. Reviewer-Petride, Sabina. (Cité pages 26 et 47.)

[6] N. Abid, S. Dal Zilio, and D. Le Botlan. A Verified Approach for Checking Real-Time Specification Patterns. In *VECoS 2012 – 6th International Workshop on Verification and Evaluation of Computer and Communication Systems*, Electronic Workshops in Computing (eWiC). BCS, Sept. 2012. (Cité page 53.)

[7] J.-R. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005. (Cité pages viii et 4.)

[8] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010. (Cité page 91.)

[9] L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 263–280, London, UK, UK, 1998. Springer-Verlag. (Cité page 41.)

[10] I. Ait-Sadoune and Y. Ait-Ameur. A proof based approach for modelling and verifying web services compositions. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. (Cité page 91.)

[11] R. Alur. Timed automata. *Theoretical Computer Science*, 126:183–235, 1999. (Cité page 22.)

[12] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, May 1993. (Cité pages xi et 6.)

[13] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993. (Cité page 152.)

[14] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc. (Cité page 13.)

[15] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. (Cité pages xi, 6, 20, 21, 40, 60 et 152.)

[16] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, Jan. 1996. (Cité pages xx, 16, 17 et 120.)

[17] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, UK, 1992. Springer-Verlag. (Cité page 16.)

[18] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. OASIS, May 2006 2006. (Cité pages xviii, 7, 83, 86, 87, 89, 94, 95, 97, 98, 99, 109, 111 et 127.)

[19] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundam. Inf.*, 40(2,3):109–124, Aug. 1999. (Cité page 39.)

[20] F. V. B. Berthomieu, P.-O. Ribet. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, Vol. 42, 2004. (Cité pages x, 6 et 28.)

[21] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. (Cité pages 37, 38, 49 et 61.)

[22] G. Behrmann, A. Cougnard, R. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Timed games for everyone. (Cité page 23.)

[23] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, FASE '00, pages 266–283, London, UK, UK, 2000. Springer-Verlag. (Cité pages 35 et 36.)

[24] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. Comparison of different semantics for time petri nets. In D. A. Peled and Y.-K. Tsay, editors, *3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005)*, volume 3707 of *Lecture*

Notes in Computer Science, pages 293–307, Taipei, Taiwan, Oct. 2005. Springer-Verlag. (Cité pages 26 et 40.)

[25] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. Comparison of the expressiveness of timed automata and time petri nets. In *Proceedings of the Third international conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'05, pages 211–225, Berlin, Heidelberg, 2005. Springer-Verlag. (Cité page 40.)

[26] B. Berard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. When are Timed Automata weakly timed bisimilar to Time Petri Nets ? In *25th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, pages 276–284, Chennai, Inde, 2005. (Cité page 40.)

[27] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0 (W3C Recommendation), Jan. 2007. (Cité page 88.)

[28] R. Bernard and et al. Altarica refinement for heterogeneous granularity models analysis, 2008. (Cité pages 39 et 66.)

[29] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008. (Cité page 26.)

[30] N. Bertrand, S. Pinchinat, and J.-B. Raclet. Refinement and consistency of timed modal specifications. In *Proc of.*, LATA '09, pages 152–163, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité page 40.)

[31] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14:25–59, March 1987. (Cité page 90.)

[32] J. Bradfield and C. Stirling. Modal mu-calculi. In *HANDBOOK OF MODAL LOGIC*, pages 721–756. Elsevier, 2007. (Cité pages 18 et 72.)

[33] V. Breugel and M. Koshkina. Models and verification of bpel franck. (Cité page 89.)

[34] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, July 1988. (Cité page 38.)

[35] T. Bultan, X. Fu, and J. Su. Analyzing conversations of web services. *IEEE Internet Computing*, 10:2006, 2006. (Cité page 90.)

[36] P. Bulychev, T. Chatain, A. David, and K. G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *Proc.*, FORMATS '09, pages 73–87, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité pages 23, 42, 43 et 78.)

[37] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag, 2004. (Cité page 15.)

[38] K. M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989. (Cité page 151.)

[39] T. Chatain, A. David, and K. G. Larsen. Playing games with timed games, 2009. (Cité page 42.)

[40] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. (Cité pages ix, 5, 18 et 35.)

[41] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 343–354, New York, NY, USA, 1992. ACM. (Cité pages x et 5.)

[42] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. (Cité pages ix et 4.)

[43] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Methodologies for specification of real-time systems using timed i/o automata. In *Proc.*, FMCO'09, pages 290–310. Springer, 2010. (Cité pages 40 et 42.)

[44] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proc. of*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM. (Cité pages 20, 22 et 23.)

[45] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, volume 600 of *Lecture Notes in Computer Science*. Springer, 1992. (Cité pages xii, 25 et 50.)

[46] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: An Appetizer. In M. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902 of *Lecture Notes in Computer Science*, chapter 3, pages 23–36. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009. (Cité pages viii et 4.)

[47] A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press. (Cité pages 89, 90, 91 et 167.)

[48] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 621–630, New York, NY, USA, 2004. ACM. (Cité page 90.)

[49] X. Fu, T. Bultan, and J. Su. Model checking xml manipulating software. *SIGSOFT Softw. Eng. Notes*, 29(4):252–262, July 2004. (Cité page 90.)

[50] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *the Proc. of 16th Int. Conf. on Computer Aided Verification (CAV*, pages 510–514. Springer, 2004. (Cité page 90.)

[51] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité pages viii et 4.)

[52] R. J. v. Glabbeek. The linear time-branching time spectrum (extended abstract). In *Proceedings of the Theories of Concurrency: Unification and Extension*, CONCUR '90, pages 278–297, London, UK, UK, 1990. Springer-Verlag. (Cité page 33.)

[53] R. J. V. Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:613–618, 1996. (Cité page 36.)

[54] A. Griffault and A. Vincent. The mec 5 model-checker. In *CAV: International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 488–491. Springer, July 2004. (Cité pages 39 et 66.)

[55] T. Henzinger. Technical report, STAN-CS-91-1380, Stanford University, 1991. title = The temporal specification and verification of real-time systems, type = Ph.D. Thesis, Technical Report, year = 1991. (Cité page 16.)

[56] T. A. Henzinger. ItŠs about time: Real-time logics reviewed. In *In Proc. CONCURŠ98*, pages 439–454. Springer, 1998. (Cité page 16.)

[57] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. *Inf. Comput.*, 173(1):64–81, Feb. 2002. (Cité page 39.)

[58] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In *REX Workshop*, pages 226–251, 1991. (Cité page 27.)

[59] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks. pages 545–558. Springer-Verlag, 1992. (Cité page 13.)

[60] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W. M. P. v. d. Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *LNCS*, pages 220–235, Nancy, France, Sept. 2005. Springer-Verlag. (Cité pages 89, 90, 91 et 98.)

[61] C. A. R. Hoare. *Communicating sequential processes (Prentice-Hall International series in computer science)*. Prentice/Hall International, April 1985. (Cité page 34.)

[62] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. (Cité pages x et 6.)

[63] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003. (Cité page 90.)

[64] A. S. Jeffrey, S. A. Schneider, and F. W. Vaandrager. A comparison of additivity axioms in timed transition systems. Technical report, Amsterdam, The Netherlands, The Netherlands, 1993. (Cité page 16.)

[65] H. E. Jensen, K. Guldstrand, and A. Skou. Scaling up uppaal: automatic verification of real-time systems using compositionality and abstraction. In *Proc. FTRTFT 2000. 84 ALTISEN ET AL*, 2000. (Cité page 41.)

[66] J. Julliand, H. Mountassir, and E. Oudot. Vesta: A tool to verify the correct integration of a component in a composite timed system. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007, Proceedings*, volume 4789 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2007. (Cité pages 42 et 143.)

[67] J. Julliand, H. Mountassir, and E. Oudot. Incremental verification of component-based timed systems. *Int. J. Comput. Appl. Technol.*, 42(2/3):159–176, Feb. 2011. (Cité pages 41, 42 et 47.)

[68] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed i/o automata. Technical report, 2003. (Cité page 40.)

[69] R. Kazhamiakin, P. Pandya, and M. Pistore. Representation, verification, and computation of timed properties in Web Service Compositions. In *Proceedings of the IEEE International Conference on Web Services*, pages 497–504, Washington, DC, USA, 2006. IEEE Computer Society. (Cité pages 90, 91 et 113.)

[70] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. Comput.*, 200(1):35–61, July 2005. (Cité page 39.)

[71] W. Khansa, J. P. Denat, and S. Collart-Dutilleul. P-time Petri nets for manufacturing systems. In *Proceedings of the International Workshop on Discrete Event Systems (WODES'96), 1996.*, pages 94–102, 1996. (Cité page 26.)

[72] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, Oct. 1990. (Cité page 16.)

[73] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997. (Cité pages x, 6, 21 et 41.)

[74] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In K. v. Hee, W. Reisig, and K. Wolf, editors, *Proceedings of the*

Workshop on Formal Approaches to Business Processes and Web Services (FABPWS'07), pages 21–35. University of Podlasie, June 2007. (Cité pages 89, 90, 91 et 98.)

[75] N. Lohmann and J. Kleine. Fully-automatic translation of open workflow net models into simple abstract bpel processes. In *In: Modellierung 2008. Volume 127 of LNI., GI*, pages 57–72, 2008. (Cité page 90.)

[76] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. D.: Analyzing interacting wsbpel processes using flexible model generation. *Data Knowl. Eng*, pages 38–54, 2008. (Cité page 90.)

[77] N. Lynch and F. Vandraager. Forward and backward simulations - part ii: Timing-based systems. *Information and Computation*, 128, 1995. (Cité pages 38 et 40.)

[78] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *in E.W. Mayr and C. Puech (Eds), Proc. STACS'95, LNCS 900*, pages 229–242. Springer, 1995. (Cité page 22.)

[79] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. (Cité pages ix et 5.)

[80] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. (Cité page 14.)

[81] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974. AAI7511026. (Cité page 26.)

[82] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd international joint conference on Artificial intelligence*, IJCAI'71, pages 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc. (Cité page 34.)

[83] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. (Cité pages viii, 4, 34, 36, 38 et 90.)

[84] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) version 2.0 part 1: Core language. Candidate recommendation, W3C, Mar. 2006. (Cité pages xviii, 7 et 84.)

[85] S. Nakajima. Model-Checking Behavioral Specification of BPEL Applications. *Electron. Notes Theor. Comput. Sci.*, 151:89–105, May 2006. (Cité pages 90 et 91.)

[86] S. Nakajima. Model-checking behavioral specification of bpel applications. *Electron. Notes Theor. Comput. Sci.*, 151(2):89–105, May 2006. (Cité page 90.)

[87] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. pages 526–548. Springer-Verlag, 1991. (Cité page 69.)

[88] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, May 2012. (Cité pages ix et 4.)

[89] D. Peled. Combining partial order reductions with on-the-fly model-checking. pages 377–390. Springer-Verlag, 1994. (Cité pages x et 5.)

[90] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct. 2003. (Cité page 86.)

[91] C. A. Petri. *Communication with Automata*. PhD thesis, UniversitÃd't Hamburg, 1962. (Cité pages viii, 4 et 24.)

[92] G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and verification of BPEL4WS. *Electron. Notes Theor. Comput. Sci.*, 151:33–52, May 2006. (Cité page 90.)

[93] Y. Qian, Y. Xu, Z. Wang, G. Pu, H. Zhu, and C. Cai. Tool Support for BPEL Verification in ActiveBPEL Engine. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 90–100, Apr. 2007. (Cité pages 90, 91 et 113.)

[94] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag. (Cité pages ix et 5.)

[95] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Cambridge, MA, USA, 1974. (Cité page 26.)

[96] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. (Cité pages viii, 4, 23 et 149.)

[97] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. *Web Services, IEEE International Conference on*, 0:43, 2004. (Cité pages 89, 90 et 91.)

[98] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The enterprise service bus: making service-oriented architecture real. *IBM Syst. J.*, 44(4):781–797, Oct. 2005. (Cité page 83.)

[99] W. Song, X. Ma, C. Ye, W. Dou, and J. Lü. Timed modeling and verification of BPEL processes using time petri nets. In *Proceedings of the 2009 Ninth International Conference on Quality Software*, QSIC '09, pages 92–97, Washington, DC, USA, 2009. IEEE Computer Society. (Cité page 90.)

[100] M. Sorea. A decidable fixpoint logic for time-outs. In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 255–271, London, UK, UK, 2002. Springer-Verlag. (Cité page 67.)

[101] C. Stahl. Transformation von bpel4ws in petrinetze. Technical report, Humboldt-Universität zu Berlin,Berlin, Germany, 2004. (Cité page 89.)

[102] C. D. Team. The Coq proof assistant reference manual, v. 8.2, 2009. (Cité pages ix, 4 et 148.)

[103] G. Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986. (Cité page 147.)

[104] S. Yovine. Kronos: A verification tool for real-time systems. (kronos user's manual release 2.2). *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997. (Cité page 21.)