THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *03/12/2014* **par :**

Iulia DRAGOMIR

Conception et vérification d'exigences de sûreté temporisées à base de contrats dans les modèles SysML

Contract-based Modeling and Verification of Timed Safety Requirements for System Design in SysML

JURY

Béatrice BÉRARD	Professeur, Université Pierre et Marie Curie	Rapportrice
Jean-Paul BODEVEIX	Professeur, Université de Toulouse	Examinateur
Susanne GRAF	Directeur de Recherche, CNRS-VERIMAG	Examinatrice
Thomas LAMBOLAIS	Maître assistant, École des Mines d'Alès	Examinateur
Alexander KNAPP	Professeur, Universität Augsburg	Rapporteur
Iulian OBER	Maître de conférences HDR, Université de Toulouse	Directeur de thèse
Christian PERCEBOIS	Professeur, Université de Toulouse	Codirecteur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse
Directeur(s) de Thèse :
Iulian OBER et *Christian PERCEBOIS*
Rapporteurs :
Béatrice BÉRARD et *Alexander KNAPP*

Abstract

Nowadays computer systems grow larger in size and more complex. Embedded in devices from different domains like avionics, aeronautics, consumer electronics, etc., they are often considered critical with respect to human life, costs and environment. A development that results in safe and reliable critical real-time embedded systems is a challenging task, considering that errors are accidentally inserted in the design. A way for system designers to tackle this issue is to use a compositional design technique based on components and driven by requirements: it allows to infer from global requirements, component properties that must locally hold.

Contract-based reasoning allows to compositionally derive correct components from global system requirements by interposing abstract and partial specifications for components. Informally, a contract models the abstract behavior a component exhibits from the point of view of the requirement to be satisfied (i.e. guarantee) in a given context (i.e. assumption). Contracts can be used to decompose and trace requirements during iterative design, but also to perform compositional verification of requirement satisfaction.

In this thesis, we present a methodology for reasoning with contracts during system design and verification within SysML. Thus, we define the syntax for contracts in UML/SysML, as well as a set of refinement relations between contracts and/or components in order to prove the system's correctness with respect to requirements. Next, we provide a formal framework that models the semantics of a UML/SysML model extended with contracts as a mapping of the language concepts to a variant of Timed Input/Output Automata. The refinement relations are formalized based on the trace inclusion relation and compositional properties are proved to hold which ensures the soundness of the methodology. The approach is instantiated for the OMEGA Profile and IFx2 toolset with partial automatic generation of proof obligations. Finally, the approach is applied on several case studies, including an industry-grade system model, which show its efficiency by comparative verification results.

Keywords: contract-based reasoning, real-time systems, safety requirements, component-based design, UML/SysML, compositional verification, Timed Input/Output Automata, model-checking

Résumé

De nos jours, les systèmes informatiques croissent en taille et en complexité. Intégrés dans des dispositifs de différents domaines tels que l'avionique, l'aéronautique, l'électronique grand public, etc., ils sont souvent considérés comme critiques à l'égard de la vie humaine, des coûts et de l'environnement. Concevoir des systèmes embarqués temps-réel critiques sûrs et fiables est une tâche difficile, étant donné que leurs modèles sont souvent source d'erreurs. Une façon pour les concepteurs de contourner cette difficulté consiste à s'appuyer sur la modélisation compositionnelle de composants logiciels pilotée par les exigences.

Le raisonnement à base de contrats permet de construire des composants sûrs à partir des exigences globales du système en interposant des spécifications abstraites et partielles entre les besoins du système et les composants eux-mêmes. Informellement, un contrat modélise le comportement abstrait d'un composant du point de vue de l'exigence à satisfaire (c.a.d garantie) dans un contexte donné (c.a.d. hypothèse). Les contrats peuvent être exploités pour décomposer et tracer les exigences au cours d'un développement itératif, mais aussi pour effectuer une vérification compositionnelle de la satisfaction des exigences.

Dans cette thèse, nous présentons une méthodologie de raisonnement à base de contrats pour la conception et la vérification de systèmes sûrs développés en SysML. Ainsi, nous définissons en UML/SysML la syntaxe des contrats et des relations de raffinement entre contrats et/ou composants qui sont utilisées pour prouver la correction du système par rapport aux exigences. Ensuite, nous proposons un cadre formel qui modélise la sémantique d'un modèle UML/SysML étendu par des contrats selon une variante d'automates temporisés entrée/sortie et nous définissons la correspondance entre ces concepts. Nous formalisons les relations de raffinement par la relation d'inclusion de traces et nous prouvons leurs propriétés compositionnelles ce qui assure la correction de la méthodologie. L'approche est instanciée pour le profil OMEGA et la boîte à outils IFx2 qui génère partiellement les obligations de preuve. Finalement, plusieurs études de cas dont une issue de l'industrie complètent la théorie pour évaluer l'approche à base de contrats et ses résultats et les comparer aux méthodes classiques de model-checking.

Mots-clés : raisonnement à base de contrats, systèmes temps-réel, exigences de sûreté, conception à base de composants, UML/SysML, vérification compositionnelle, automate temporisé entrée/sortie, model-checking

Acknowledgements

Doing a PhD was a long-date dream and it would not have been possible without the support and help of a long list of special persons.

Foremost, I would like to thank my two advisers, Iulian Ober and Christian Percebois, for all they taught me during this time. My collaboration with Iulian started 5 and a half years ago, during the internships of my master and crowned by this thesis. It was an honor for me to work with him for such a long time. Words cannot express how grateful I am for all the opportunities he gave me, in the everyday life of research, teaching and administration duties. I thank him for his guidance whenever I diverged from "right" path, for his patience in teaching me – only he knows how difficult that must had been –, spontaneous answers to my numerous questions, availability and all the qualities that are too plentiful to enumerate here and which make of him a great mentor. Most importantly, I am grateful for the trust he showed me. I will always be in debt to you!

I am very grateful to Christian for embarking in this adventure and bringing a fresh perspective on our work, and not only. I appreciate his dedication, gentleness, pedagogy, honesty, criticism and pertinent remarks. Working with Christian was a great and enriching human experience that I will never forget. Thank you for being a role model for me!

I would like to thank Béatrice Bérard, Jean-Paul Bodeveix, Susanne Graf, Alexander Knapp and Thomas Lambolais for making me the honor and accepting to be on my defense committee. I thank Jean-Paul for presiding this committee and for his gentleness and support. I apologize to my two reviewers, Béatrice and Alexander, for making them suffer with such a long thesis. I appreciate your hard work and the effort you put into improving this manuscript. I thank Susanne and Thomas for their interest in this thesis and all the relevant comments in improving our work and perspectives. A special thought goes to Susanne, whom I know since my master internship and who took the time to discuss this research and offer valuable advice each time we met.

This research idea originated in the "Full Model Driven Development for On-Board Software" project. I am obliged to all my colleagues, Éric Conquet, David Lesens,

François-Xavier Dormoy, Marius Bozga, Susanne Graf, and those who I forgot to mention here, for their direct and indirect contribution to this thesis. Special thanks to David for being the promoter of this research, our rich discussions and his aid.

This work would have not existed without the endorsement of my previous teachers, Crina Grosan and Dan Chiorean from the "Babes-Bolyai" University, Faculty of Mathematics and Computer Science. I wish to thank Crina for rooting the idea of making a PhD and believing that I have what it takes to do research. I thank Dan for his encouragements, his genuine trust in my abilities and teaching me how to be critical with my work and always ask for more.

The period spent as a doctoral student had also its fun moments. I am grateful to the MACAO members, Ileana, Hervé, Jean-Michel, Hanh Nhi, Sophie, Thierry, Bernard, Brahim, for making a place for me in their group, their concern about my well being and the good moments we spent together at lunches, coffee breaks, seminars, etc. A special mention for Hervé who mentored my teaching and for Ileana who involuntary upheld the role of a mom while I was away from home. I thank the ACADIE members, Mamoun, Jean-Paul, Ralph, Erik, Jean-Baptiste, Martin, Jan, for considering me one of their own, our discussions more or less research-related and their recommendations. I thank my colleagues from the Computer Science department of IUT A and Faculté des sciences et d'ingénierie for welcoming me in their structures and allowing to flourish as a teacher.

Finally, I thank my fellows, Nadia, Elie, Wilmer, Selma, Bertrand, Manzoor, Jacob, El Arbi, Adel, Rahma, Hajer, Faten, ..., for the moments spent together and their friendship. I also thank to my old and new friends, Monica, Ancuta, Samira, Madalina, Dana, Valentin, Ana-Maria, Adrian, Tomasz, Christophe, Alexandra, ..., for their continuous encouragements. Sorry for those that I forgot to mention here. Things would have not been the same without you.

Last but not least I am grateful to my family for their faith in me, even though it was not always clear why and what I kept studying. My achievement is mainly due to my parents and their lessons to stay strong, believe in yourself and pursue the dream no matter what. I am happy and honored to be their daughter and I hope that I will continue making them proud of me. I am thankful to Rémy for putting up with me day by day, providing helpful advices and believing in me maybe more than I do in myself. I am lucky for you to be a part of my life.

Toulouse, December 9, 2014 Iulia Dragomir

Contents

List of Figures

List of Tables

Résumé étendu

1 Introduction

La conception de systèmes critiques sûrs et fiables est difficile, comme plusieurs catastrophes récentes le montrent. Par example, le vol 501 d'Ariane 5 a explosé 37 secondes après le lancement en raison d'une conversion de données incorrecte causant la perte de plus de 370 millions de dollars, le système anti-balistique MIM-104 Patriot n'a pas réussi à intercepter un missile causant la mort de 28 soldats en raison d'une dérive de l'horloge interne qui s'est soldée par un mauvais calcul pour la recherche et le suivi du projectile, et la machine à rayonnement Cobalt-60 produite par Multidata Systems a mis en danger plusieurs dizaines de patients en 2000 à cause de mauvais calculs dépendant de la séquence d'entrée des données. Ces erreurs (ou bugs) sont responsables de dommages importants relatifs à la sécurité, à l'environnement et même à la perte de vies humaines. Assurer la sûreté et la correction du comportement de ces systèmes critiques est essentiel. Un cas particulier est représenté par les systèmes temps réel pour qui, en plus de l'ordre d'actions ou de valeurs, la correction repose également sur le moment où une action doit être effectuée.

Le développement de systèmes embarqués temps-réel critiques est une tâche difficile. Il y a deux principaux facteurs qui doivent être pris en compte lors du développement : (1) quelle est la meilleure méthode pour concevoir le système avec des efforts et coûts minimes et (2) comment peut-on s'assurer que le système conçu est correct par rapport aux exigences requises. En effet, les systèmes croissent en taille et en complexité, ainsi que le nombre d'erreurs qu'ils contiennent et qui deviennent plus difficiles à identifier et corriger, alors que leur impact sur le produit final peut avoir des conséquences catastrophiques.

Afin de maîtriser la taille du système, les ingénieurs ont adopté une méthode de conception compositionnelle basée sur les composants et qui permet de décomposer de manière récursive le problème à résoudre, généralement une exigence, jusqu'à ce que le niveau de granularité souhaité soit atteint. En conséquence, les concepteurs vont travailler sur de plus petites spécifications qui sont plus faciles à développer, appelées composants atomiques, ou sur l'assemblage de composants par composition qui se traduit par un composant hiérarchique. Cette méthode de conception présente plusieurs avantages : séparation des préoccupations dans la décomposition du système, développement incrémental par raffinements successifs, implémentation de composants par plusieurs et différentes équipes d'ingénieurs et réutilisation de composants.

Cependant, avoir plusieurs fournisseurs développant des systèmes intégrés fondés sur des exigences communes entraîne un risque d'erreur en raison de la difficulté à décomposer les exigences globaux sur des composants, mais aussi la mauvaise interprétation des exigences du système allouées au logiciel. Par conséquent, la limite de cette méthode réside dans l'aspect compositionnel inhérent : il est difficile de concevoir un réseau de composants qui satisfont par leur interaction une exigence globale, alors que les composants sont généralement impliqués dans la satisfaction de plusieurs besoins.

En ce qui concerne la deuxième question exprimée plus haut, nous devons prendre en compte le fait que les premiers modèles sont souvent réalisés en utilisant des langages semi-formels tels que UML [91], SysML [90] ou AADL [148], qui manquent de mécanismes pour formaliser des exigences et prouver leur satisfaction. Les erreurs potentiellement introduites au cours du développement sont généralement découvertes tardivement et par des processus très couteux. Par conséquent, les dernières décennies ont vu une utilisation accélérée des techniques de vérification et validation dès les premières phases de développement afin de garantir le plus rapidement possible la correction de la conception, et ainsi réduire les coûts de production et augmenter la qualité des systèmes. D'un modèle système correct via des relations de raffinement prouvées, nous pouvons obtenir une implémentation correcte du système qui peut être déployée immédiatement. Ceci implique que les méthodes formelles peuvent être utilisées pour obtenir des implémentations correctes par construction à partir de spécifications de haut niveau si elles sont intégrées dans un processus de développement itératif.

Les modèles ainsi conçus sont validés à l'aide d'un assortiment de techniques, y compris les revues de conception [132], les tests, la simulation interactive et le *model-checking* [141, 49, 52]. Les trois premières méthodes permettent de détecter des erreurs de façon légère car elles explorent seulement un sous-ensemble des comportements du système et, en conséquence, elles ne garantissent pas la correction du système par rapport aux exigences. Le model-checking est une technique entièrement automatisée qui explore de façon exhaustive les comportements du système représentés par un modèle d'espace d'états. L'espace d'états est généralement un graphe fini. Cependant, pour de très grands systèmes, l'espace d'états ne peut pas être complètement calculé, ce qui rend le model-checking limité par le problème d'explosion de l'espace d'états : le système devient vite inextricable si les composants s'exécutent en parallèle, car le nombre d'états croît de façon exponentielle directement lié au nombre de composants. Par conséquent, un verdict pour la satisfaction des besoins ne peut pas être fourni. Plusieurs exemples [71, 40, 21] montrent que les techniques de vérification actuelles sont impuissantes face à la complexité des systèmes industriels.

Trois types d'optimisation ont été étudiées dans la littérature :

1. réduire l'espace d'états en modifiant la représentation mathématique,
2. modéliser des abstractions pour les composants du système et vérifier que l'exigence est satisfaite par le modèle abstrait, et
3. décomposer l'exigence globale en plusieurs besoins qui doivent être satisfaits localement par les composants, ce qui correspond à une approche compositionnelle.

Nous considérons que la première optimisation est partielle, si elle ne peut pas suffisamment réduire la complexité des modèles système afin de les rendre vérifiables. Les deux autres approches sont duales, car la spécification joue le rôle d'une abstraction du comportement du composant pour la deuxième, et une contrainte sur le comportement du composant pour la troisième. Cependant, leur principale limite réside dans le fait qu'elles ne peuvent pas prendre en compte le comportement de l'environnement et, par conséquent, le composant doit correctement raffiner son abstraction/exigences indépendamment, c.à.d dans tous les environnements. Cela peut être difficile à prouver car entre un composant et son environnement il y a généralement des dépendances mutuelles sur lesquelles se fonde leur correction. En conséquence, l'environnement, qui peut être une source pour l'explosion de l'espace d'états, doit être pris en compte par le raffinement.

Dans cette thèse nous proposons de combiner les techniques d'abstraction et de composition et de définir une spécification unique pour un composant qui soit à la fois abstraite et partielle. L'environnement doit également être contraint par une telle spécification. En conséquence, nous utilisons la notion de *contrat* pour un composant, défini par une paire (hypothèse, garantie) où l'hypothèse est une abstraction du comportement de l'environnement et la garantie est une abstraction du comportement du composant par rapport à l'exigence à satisfaire, étant donné que l'environnement se comporte comme l'hypothèse. L'hypothèse est correcte si l'environnement la raffine dans le contexte abstrait de la garantie. Ce type de raisonnement est circulaire car la correction du composant et de l'environnement tient compte du comportement abstrait de l'autre, et peut être prouvé correct dans certains cas.

Informellement, un contrat modélise le point de vue d'un composant et sa contribution dans la satisfaction d'une exigence. En conséquence, un composant peut implémenter plusieurs contrats, un pour chaque exigence à satisfaire. L'ensemble des contrats correspondant au réseau de composants doit s'assembler correctement et satisfaire l'exigence. Le nombre de relations à vérifier pour que le raisonnement à base de contrats soit correct est linéaire au nombre de composants modélisés et nous pouvons supposer que les compositions concernées sont en général réduites et peuvent être traitées par des outils de vérification automatiques.

Les contrats représentent un atout précieux dans la conception correcte par construction de modèles à base de composants, car ils peuvent être utilisés pour :

1. contraindre le comportement d'un composant par rapport à une exigence, alors que plusieurs contrats peuvent être intégrés dans la même implémentation,
2. substituer et réutiliser des composants, éventuellement conçus auparavant, qui satisfont le contrat donné,
3. implémenter indépendamment des composants en se basant sur le contrat donné sans contester la satisfaction des besoins, et
4. concevoir de façon itérative des systèmes en utilisant des relations de raffinement prouvées.

Ces propriétés sont prises en charge par les trois relation de raffinement dont un contrat peut faire l'objet : la *conformité* vérifie si un contrat satisfait une exigence, la *dominance* vérifie le raffinement entre contrats et l'*implémentation* vérifie si un composant satisfait son contrat. En outre, le raisonnement à base de contrats offre diverses possibilités : le mapping et le suivi des exigences vers des composants, l'évolution des besoins au cours du développement, l'aide à des revues de modèles, l'intégration virtuelle de composants et, surtout, la vérification compositionnelle.

Les notions liées aux contrats décrites ci-dessus ont été définies dans [143, 144, 142] sous la forme d'une *méta-théorie*. Par méta-théorie, nous désignons un cadre générique de contrats qui décrit comment le raisonnement peut être appliqué pour la conception de systèmes et leur vérification compositionnelle, sans pour autant fournir une définition précise de ces concepts. Afin d'obtenir un cadre de travail pour un modèle de composants spécifique, il faut formaliser le cadre des composants

— définir au moins les concepts de composant, composition et raffinement — et le cadre des contrats — définir la conformité, la dominance et l'implémentation. En plus, un ensemble de propriétés compositionnelles doit être prouvé dans le cours de cette instanciation afin de garantir la correction du raisonnement.

Contribution

Malgré les avantages évidents, l'ingénierie système n'utilise pas le raisonnement à base de contrats comme méthode de développement de modèles système décrits avec par langages semi-formels (e.g. UML, SysML, etc.) en raison de l'absence d'une définition d'un cadre correct et complet axé sur les contrats directement applicable à de telles conceptions. L'objectif de cette thèse est de greffer le raisonnement à base de contrats dans le processus de conception à base de composants et de vérification des exigences pour des systèmes embarqués temps-réel critiques décrits en SysML. A notre connaissance, cette étude est la première à relier les langages de modélisation de haut niveau et les contrats comportementaux formels.

Notre contribution est multiple. Afin d'utiliser les contrats comme des éléments de première classe en SysML, nous définissons la syntaxe des notions liées aux contrats par un méta-modèle basé sur UML. Un ensemble de règles de bonne formation est défini sur le méta-modèle afin d'assurer sa conformité à la méthode de raisonnement à base de contrats décrite dans [143, 144, 142]. Par exemple, ces règles couvrent les actions qu'un contrat/composant peut exécuter et leur raffinement en intégrant plusieurs contraintes dans le même composant. Nous instancions le méta-modèle dans le contexte du profil OMEGA de manière à pouvoir l'utiliser dans des éditeurs de modèles. OMEGA [87] est un profil UML/SysML, qui permet de développer de manière rigoureuse des systèmes temps-réel en offrant une liaison à la boîte à outils de vérification et validation IFx [34].

Deuxièmement, nous formalisons la sémantique du langage à base de composants SysML étendu avec des contrats par une variante des automates temporisés entrée/-sortie (TIOA). Cette transformation rend les relations de raffinement modélisées entre les contrats et/ou composants vérifiables, tandis que leur satisfaction implique la satisfaction de l'exigence globale. Par conséquent, nous définissons la correspondance de concepts de SysML à TIOA et nous esquissons la façon d'explorer un modèle afin de générer les obligations de preuve qui correspondent aux relations de raffinement modélisées. Nous implémentons partiellement cette transformation dans la boîte à outils IFx2 par un compilateur, qui prend en entrée un modèle système OMEGA dans le format XMI 2.0 et qui produit le réseau de TIOA sur lequel la vérification sera effectuée. La version actuelle du compilateur étend les fonctionnalités de IFx2, c.a.d simulation et model-checking de systèmes temps-réel, à des modèles système compatibles avec UML 2.3 / SysML 1.1.

Nous construisons sur le cadre formel de composants la théorie à base de contrats en définissant une obligation de preuve pour chaque type de relation de raffinement. Le framework à base de contrats obtenu est une instanciation de la méta-théorie définie dans [143, 144, 142], dont les composants sont représentés par des TIOA et les obligations de preuve par une relation d'inclusion de traces temporisées qui prend en compte l'environnement et qui est nomée ci-après raffinement dans un contexte. Nous prouvons que le raffinement dans un contexte est préservé par la composition et garantit la correction du raisonnement circulaire, deux résultats importants qui constituent une condition préalable de la méta-théorie pour assurer la correction de la méthode. Comme l'inclusion de traces temporisées n'est pas décidable ce qui implique qu'elle ne peut pas être vérifiee automatiquement, nous proposons d'utiliser le model-checking sur chaque obligation de preuve en transformant l'exigence locale en une exigence de sûreté temporisée et déterministe formelle. Informellement, une exigence de sûreté modélise que quelque chose de mauvais (c.à.d inattendu) n'arrive jamais pendant l'exécution du système. Nous prouvons que cette transformation et l'application de l'analyse d'accessibilité sont suffisantes pour garantir l'inclusion de traces temporisées. Le cadre formel et la méthode de vérification imposent certaines restrictions à l'égard de l'expressivité des contrats : nous discutons leur impact sur le langage de composants utilisé dans la modélisation de contrats et nous définissons une sémantique temporisée particulière qui satisfait par défaut les restrictions nécessaires, qui imposent qu'une garantie doit être une exigence de sûreté temporisée déterministe.

Enfin, nous évaluons l'approche sur deux études de cas dont une est issue de l'industrie, le *Solar Generation Wing System* (SGS) de l'*Automated Transfer Vehicle* (ATV), pour lesquelles nous vérifions la satisfaction d'une exigence de sûreté globale. Les résultats obtenus sont encourageants : pour l'étude de cas portant sur le SGS, l'approche à base de contrats a nécessité l'effort de 5 personnes*jours pour modéliser les contrats et effectuer la vérification, tandis que le model-checking monolithique, même en conjonction avec des techniques de réduction, ne fournit pas de résultat.

Organisation

Cette thèse est structurée en trois parties. La première met l'accent sur la motivation et le contexte de notre travail en décrivant les limites des techniques actuelles de conception et vérification. La deuxième partie présente la contribution théorique de notre travail, l'instanciation et la mise en œuvre de la méta-théorie à base de contrats sélectionnée pour des systèmes en SysML. La dernière partie décrit la contribution pratique et évalue les résultats de vérification de notre méthode par rapport au model-checking monolithique.

2 Raisonnement à base de contrats pour les systèmes hiérarchiques à base de composants

La conception à base de contrats est un paradigme de développement émergeant de la conception et de la vérification modulaire et compositionnelle des systèmes, qui a ses racines dans la représentation axiomatique des programmes [104, 2].

Le principe est de définir une spécification partielle et abstraite sous la forme d'un contrat que chaque composant doit mettre en œuvre. La spécification est partielle car elle n'est définie que par rapport à une exigence et abstraite car elle ne fournit pas de détails d'implémentation des exigences. En conséquence, une spécification représente la projection d'une exigence sur le composant qui doit la satisfaire. En ce qui concerne la conception du système, les contrats sont un atout précieux car ils permettent : la substitution des composants et leur réutilisation, le développement incrémental et l'implémentation indépendante des composants. Complété par une approche formelle, le processus de conception peut entraîner des implémentations correctes par construction. Outre la conception, les contrats peuvent être utilisés pour la décomposition et le suivi des exigences sur les composants, l'intégration virtuelle des composants [60] et, surtout, la vérification compositionnelle.

Des travaux récents ont exploré la notion de contrat en vue de définir une démarche pour le développement de systèmes en termes de méta-théories, mais aussi de fournir directement des théories et des outils pour formalismes à base de composants spécifiques. Rappelons que par méta-théorie nous entendons un cadre générique, indépendant d'un formalisme de spécification particulier des composants. Pour obtenir une théorie concrète, nous devons formaliser ces composants et contrats et leurs relations de raffinement, tout en prouvant des résultats de compositionalité. Dans ce qui suit, nous décrivons les (méta)-théories à base de contrats définies dans la littérature, pour lesquelles nous passons en revue les notions de base.

2.1 Une méta-théorie pour le raisonnement à base de contrats

Dans la suite, nous présentons la méta-théorie proposée dans [143, 144, 142] qui constitue la base de notre travail. Nous décrivons les concepts clés et les propriétés qui permettent de raisonner avec des contrats de façon compositionnelle pour les systèmes hiérarchiques. Un atout important de cette méta-théorie est la méthodologie qu'elle définit, illustrée par la figure 3.1, et qui est expliquée ci-dessous.

Supposons, à n'importe quel niveau de la décomposition hiérarchique d'un système, un sous-système S obtenu par la composition de plusieurs composants $K_1, K_2, \ldots, K_n$ sur lequel nous voulons prouver la satisfaction d'une exigence φ. Cette méta-théorie laisse libre le choix de l'opérateur de composition à utiliser. Afin de simplifier la présentation, nous supposons l'existence de la notion de compatibilité entre les composants et l'existence d'un opérateur de composition pour chaque paire de composants compatibles, noté $\|$, qui est unique et associatif. En conséquence, S est donné par la composition $K_1 \parallel K_2 \parallel \ldots \parallel K_n$.

Afin d'utiliser la méthodologie, nous commençons par modéliser un *contrat* pour chaque composant que le sous-système contient.

Définition 1 (Contrat). Un contrat $\mathcal{C}$ est une paire de composants compatibles (A, G) dont le composant A est appelé *hypothèse* et le composant G est appelé *garantie.*

Un contrat $\mathcal{C}_i = (A_i, G_i)$ pour le composant K_i est un modèle abstrait de l'implication du composant dans la satisfaction de l'exigence φ. L'hypothèse modélise le comportement de l'environnement de K_i et la garantie modélise le comportement attendu pour K_i si l'environnement se comporte tel que l'hypothèse. Plusieurs contrats peuvent être spécifiés pour le même composant, surtout lorsque la satisfaction de plusieurs propriétés du système S doit être prouvée. L'exemple de la figure 3.1 présente un sous-système S formé de trois composants K_1, K_2 et K_3 et de l'environnement E avec lequel S communique. Pour chacun de ces composants, un contrat C_i est modélisé.

Afin de prouver la satisfaction de φ, le raisonnement se poursuit en vérifiant que chaque composant *satisfait* son contrat, noté $K_i \models \mathcal{C}_i$. Pour définir la satisfaction d'un contrat, la méta-théorie se base sur l'existence d'un opérateur de *raffinement dans un contexte.* Cet opérateur entre deux composants K_i et K_j dans un environnement E, noté $K_i \sqsubseteq_E K_j$, modélise informellement que le composant K_i composé avec E se comporte de façon similaire à K_j composé avec E. Même si le raffinement dans un contexte n'est pas précisément défini, la méta-théorie requiert que cette relation soit compositionnelle et qu'elle garantit la correction du raisonnement circulaire ; ces propriétés sont nécessaires pour prouver le théorème 1 ci-dessous.

Basée sur le raffinement dans un contexte, la relation de satisfaction d'un contrat est définie comme suit :

Définition 2 (Satisfaction d'un contrat). Soit $\mathcal{C} = (A, G)$ un contrat pour le composant K. Alors K *satisfait* le contrat $\mathcal{C}$, noté $K \models \mathcal{C}$, si et seulement si $K \sqsubseteq_A G$.

Nous remarquons que la méta-théorie n'impose aucune contrainte par rapport à la signature des composants K, A et G. Par exemple, dans notre instance de la méta-théorie, nous définissons A et G sur un sous-ensemble — signature et comportement — de K. Ceci nous permet de conserver la spécification d'un contrat abstraite, où seule l'information essentielle concernant l'exigence à satisfaire est considérée.

La deuxième étape du raisonnement illustré à la figure 3.1 consiste à définir un contrat $\mathcal{C} = (A, G)$ pour le sous-système S et prouver que l'ensemble des contrats $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *domine* le contrat $\mathcal{C}$.

Définition 3 (Dominance). Un ensemble de contrats $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *domine* un contrat $\mathcal{C}$ si et seulement si pour tout ensemble de composants $\{K_i\}_{i=\overline{1,n}}$ les conditions suivantes sont satisfaites: $K_i \models \mathcal{C}_i$, $i = \overline{1,n} \implies K_1 \parallel K_2 \parallel \ldots \parallel K_n \models \mathcal{C}$.

La relation de dominance ainsi définie utilise la composition de composants afin d'éviter la définition d'un opérateur de composition de contrats. Pourtant, lors de l'utilisation du raisonnement à base de contrats pour la vérification, nous ne sommes pas intéressés à manipuler des implémentations qui sont la principale cause de l'explosion de l'espace d'états dans les grands systèmes afin d'établir la dominance. Cette méta-théorie fournit un résultat alternatif pour prouver la dominance qui revient à vérifier un ensemble de relations de satisfaction de contrats, à condition que le raffinement dans un contexte soit compositionnel et garantisse la correction du raisonnement circulaire.

Théorème 1 (Condition suffisante pour la dominance). *Si le raffinement dans un contexte est compositionnel et garantit la correction du raisonnement circulaire, alors pour prouver que $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ domine $\mathcal{C}$, il est suffisant de prouver que*

$$\begin{cases} G_1 \parallel ... \parallel G_n \models \mathcal{C}, \text{and} \\ A \parallel G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n \models \mathcal{C}_i^{-1}, \ \forall i \in \overline{1,n} \end{cases}$$

où $\mathcal{C}_i^{-1} = (G_i, A_i)$ *dénote le contrat inverse de* $\mathcal{C}_i$*.*

La première condition modélise le raffinement d'une garantie plus abstraite par un ensemble de garanties plus concrètes. La deuxième condition exprime que les hypothèses individuelles doivent être raffinées par les garanties des autres composants avec l'hypothèse globale sur l'environnement.

Comme les systèmes réels se traduisent souvent par une architecture en plusieurs couches, l'étape de dominance peut être itérée jusqu'à atteindre le contrat du composant pour lequel la propriété d'intérêt φ est définie. Pour simplifier la présentation, la figure 3.1 montre une seule étape de dominance.

Afin de prouver la satisfaction d'une exigence φ, nous devons nous assurer que l'hypothèse A modélisée pour l'environnement E dans le contrat global $\mathcal{C}$ est correcte. La troisième étape de la méthodologie consiste à vérifier la satisfaction du contrat inverse $\mathcal{C}^{-1}$. Nous notons que cette étape est nécessaire quand le système à vérifier S est un système ouvert, c.à.d lorsqu'il interagit avec un environnement. Si l'exigence φ est exprimée sur un système fermé, alors l'hypothèse globale n'est pas à modéliser et cette étape peut être ignorée.

La dernière étape du raisonnement consiste à prouver que le contrat global *est conforme* à l'exigence φ. La relation à vérifier à ce stade est $A \parallel G \preceq \varphi$, où $\preceq$ désigne un opérateur de conformité laissé ouvert dans la méta-théorie [143, 144, 142]. Pour simplifier les choses, notre instance utilise le même formalisme pour spécifier les exigences et les composants et la relation de raffinement (dans un contexte vide car $A \parallel G$ est un système fermé) comme relation de conformité.

La stratégie détaillée ci-dessus est présentée de manière ascendante et suppose que le système a déjà été conçu ; donc le concepteur utilise les contrats pour prouver la satisfaction d'une exigence. Cette méthode peut également être appliquée d'une manière descendante afin de concevoir des systèmes corrects vis-à-vis des exigences à satisfaire.

2.2 Travaux connexes

Des méta-théories à base de contrats ont été développées également pour les théories de spécifications. Une théorie de spécifications est une algèbre complète qui, outre les opérateurs de composition et de raffinement, définit un opérateur de conjonction logique qui permet de dériver un composant satisfaisant deux spécifications différentes et un opérateur de quotient qui permet de calculer à partir d'une spécification du système partiellement implémentée la spécification la plus grossière de la partie restante (non implémentée). Par conséquent, le but d'une théorie de spécifications est de fournir des résultats de substitution permettant une conception compositionelle, alors que son utilisation dans la vérification compositionelle n'a pas reçu autant d'attention. En revanche, la méta-théorie que nous utilisons fournit l'ensemble minimal d'opérateurs nécessaires pour la conception et la vérification, même si elle pourrait subir un surcoût important, en particulier lors de la conception. La conjonction logique et le quotient peuvent être formalisés et ajoutés à notre théorie ; leur utilisation doit être indépendante du raisonnement décrit ci-dessus suffisant pour la vérification.

La méta-théorie de [17] est construite sur une théorie de spécifications et est similaire à [143, 144, 142]. Les différences principales portent sur (1) la spécification des contrats et (2) la méthode de raisonnement avec les contrats. En ce qui concerne le premier point, la théorie de [17] ne prend pas en charge le raffinement de signature, à savoir la capacité d'un contrat de se concentrer seulement sur quelques-unes des entrées/sorties d'un composant en faisant abstraction des autres, ce qui est explicitement traité par notre théorie. Une solution partielle est présentée dans [18] où les contrats sont définis sur un sous-ensemble de la signature du composant. Cependant, [18] ne permet pas de raisonner séparément avec les contrats : à chaque étape de la conception, une spécification consiste en un composant et un ensemble de contrats sur les ports tel que l'union de leurs signatures soit égale à la signature du composant. En outre, le raffinement de signature entre les spécifications est interdit car les définitions des relations de raffinement entre composants et contrats demandent que les membres des spécifications aient la même signature.

En ce qui concerne la méthode de raisonnement avec les contrats, les méta-théories de [17, 18] souscrivent au raisonnement hypothèse/garantie, c.à.d le raffinement de l'environnement doit être correct dans n'importe quel contexte, formellement écrit $E \sqsubseteq A$. Pour prouver la dominance, la méta-théorie de [17] fait usage d'un opérateur de composition de contrats, qui peut être utilisé pour calculer le contrat le plus "fort" $\mathcal{C}_1 \boxtimes \mathcal{C}_2$ satisfait par la composition de deux composants qui satisfont $\mathcal{C}_1$ et $\mathcal{C}_2$. L'étape de dominance est réduite à l'obligation de preuve suivante: $\mathcal{C}_1 \boxtimes \mathcal{C}_2 \boxtimes \ldots \boxtimes \mathcal{C}_n \preceq \mathcal{C}$, où $\mathcal{C}$ est le contrat dominé. Toutefois, la composition de contrats est partielle ; elle ne peut pas être définie pour certaines paires de contrats, car elle est basée sur le quotient qui est lui-même partiel. La méta-théorie de [18] utilise également un opérateur de composition des spécifications qui dans ce cas compose les contrats sur les ports et les composants contenus. Ainsi, l'utilisation de ces opérateurs peut se révéler difficile pour les systèmes vastes et complexes. En outre, dans les deux méta-théories, l'échec à prouver la dominance n'est pas traité et on ignore dès lors si l'utilisateur pourrait en extraire un contre-exemple, en identifier la cause et la corriger. En revanche, la méta-théorie de [143, 144, 142] permet de déduire, à partir des obligations de preuve qui constituent les conditions suffisantes pour la dominance, quel contrat est incorrect. Dans le cas de notre instance avec des automates temporisés entrée/sortie, il est possible de dériver un contre-exemple en cas d'insatisfaction d'une exigence. Sur une note plus générale, les méta-théories de [17] et [18] ne décrivent pas explicitement comment une exigence du système doit être formalisée et comment sa satisfaction peut être prouvée avec des contrats. Nous considérons que la méthodologie de raisonnement présentée dans [143, 144, 142] et décrite précédemment est un atout important dans l'application de la méta-théorie à des domaines concrets.

Le raisonnement hypothèse/garantie est une ligne de recherche de longue date, bien que les approches classiques considèrent les spécifications logiques [53, 95, 2]. L'approche la plus récente [42, 43] représente ces spécifications sous la forme d'ensembles de traces entrées/sorties et gère le raffinement de la signature dans la satisfaction d'un contrat mais seulement pour des spécifications non-temporisées. Travailler directement sur les traces sans une spécification opérationnelle intermédiaire est délicat car les équipes d'ingénieurs ne sont pas nécessairement sensibilisées à la formalisation du comportement. En outre, l'approche de raisonnement est similaire à celle proposée dans [17], en particulier en ce qui concerne la composition des contrats. En tant que tel, les remarques formulées ci-dessus par rapport à la méthodologie de raisonnement restent valables pour [42].

Les théories d'interfaces [65, 66, 67, 117, 39, 44] sont également liés aux contrats car elles peuvent être utilisées pour exprimer les hypothèses et les garanties d'un composant, mais ils le font dans la même spécification. Une interface englobe l'hypothèse représentée par ses entrées et la garantie représentée par ses sorties, et décrit comment un composant et son environnement interagissent. En général, la notion de contrat qui fusionne l'hypothèse et la garantie est bien adaptée pour obtenir des environnements compatibles dans lesquels les composants peuvent travailler conjointement. En fait, une interface joue le même rôle que la composition entre une hypothèse et une garantie tels qu'elles sont précédemment présentées et qui doit être manuellement calculée par le concepteur. Garder l'hypothèse et la garantie séparées permet : (1) de modéliser les propriétés des composants par des automates indépendants et (2) d'effectuer et vérifier le raffinement de l'hypothèse ou de la garantie, ainsi que du composant, indépendamment.

3 Modélisation de contrats comportementaux hypothèse/garantie en SysML

Afin d'exploiter le raisonnement à base de contrats dans une approche de développement guidé par le modèle, nous proposons ici une extension des langages de modélisation en introduisant la notion de contrat et les relations de vérification nécessaires. Dans la suite, nous décrivons les notions liées aux contrats présentées dans la section 2.1 par un méta-modèle. Un ensemble de règles de bonne formation est défini et formalisé sur ces concepts afin d'assurer la conformité d'un modèle étendu par des contrats à la méta-théorie. Cette présentation est faite pour UML ; l'extension à SysML étant directe en utilisant les méta-classes correspondantes du paquetage *UML4SysML* [90].

3.1 Le contexte de modélisation SysML

Un système à base de composants, tel que présenté à la figure 3.1, est modélisé du point de vue architectural en *UML4SysML* par deux notions: la *classe* qui permet de définir des types à utiliser dans le modèle et la *structure composite* qui permet de faire face à la complexité croissante des systèmes en décrivant les instances de classes et la façon dont elles sont composées et reliées dans une structure hiérarchique.

Dans la suite, nous supposons que le lecteur est familier avec les notions de classe, de structure composite et tous les autres éléments de modélisation pertinents de UML et SysML. Ces notions sont décrites plus en détail dans la section 4.1. Nous imposons certaines contraintes quant à la façon dont ces éléments de modélisation doivent être utilisés, afin d'éviter toute ambiguïté dans les modèles, mais aussi afin d'assurer un typage statique rigoureux des structures composites et leur définir une sémantique opérationnelle précise. Nous résumons ici ces contraintes, elles sont décrites en détail et justifiées dans la section 4.1. Les composants peuvent communiquer seulement par des signaux asynchrones, envoyés et reçus via des ports. Le type d'un port consiste en une interface qui ne définit que les réceptions de signaux capables à traverser les ports. Un port peut être un point d'entrée pour un composant si l'interface est fournie ou un point de sortie si l'interface est requise. La signature d'un composant est donnée par l'ensemble des réceptions des signaux définies pour tous les ports d'un composant. Dans une structure composite, les ports de différents composants qui doivent communiquer sont connectés par des connecteurs. Un connecteur peut être modélisé entre deux ports seulement s'ils ont le même type. De plus, un connecteur est correctement modélisé entre deux composants au même niveau hiérarchique si les ports ont des directions disjointes et entre un composant et la structure composite qui le contient si les ports ont la même direction. Les connecteurs qui simulent une communication de type multicast (c.à.d un port de sortie est connecté à deux ports entrants différents) sont interdits car ils peuvent induire un surcout important dans la sémantique sous-jacente et, plus encore, sont rarement utilisés dans la conception de systèmes critiques. Cependant, ce type de communication peut être réalisé par une modélisation explicite de signaux distincts envoyés par des ports différents à leurs cibles initiales.

Le comportement d'un composant atomique est modélisé par une machine à états. Pour décrire des actions (c.à.d l'effet des transitions), nous utilisons un sous-ensemble du langage d'actions formalisé par fUML [130] constitué de l'envoi d'un signal, de l'affectation des attributs et de l'évaluation des expressions. Le comportement d'un composant composé est donné par l'exécution parallèle de ses sous-composants.

Puisque nous cherchons à modéliser des systèmes temps-réel, nous devons étendre l'ensemble des notions utilisées pour décrire le comportement des composants avec des concepts temporisés. Pour cela, nous choisissons d'utiliser le profil OMEGA UML/SysML [87] dédié à la conception formelle et la vérification de systèmes embarqués temps-réel grâce à la boîte à outils IFx [34]. Ce profil, qui représente notre cadre de travail, garantit les contraintes précédemment présentées en terme de règles de bonne formation et en conséquence définit une sémantique opérationnelle précise et cohérente adaptée aux techniques formelles de vérification.

Concernant le comportement temporisé, OMEGA offre une extension inspirée par les automates temporisés avec urgence [31] : le profil contient la notion de *Timer* (horloge) et des stéréotypes correspondant aux urgences sur les transitions sortantes qui permettent de contrôler l'écoulement du temps dans un état. Le *Timer* permet de mesurer la durée et peut être réglé et remis à zero par deux fonctions : *set()* et *reset()*. Ces horloges peuvent être facilement utilisées pour décrire des gardes temporisées. Concernant les stéréotypes, nous utilisons par la suite «*eager*» qui modélise le fait que la transition doit être exécutée dès son activation (c.à.d l'écoulement du temps est désactivé) et «*lazy*» qui modélise le fait que la transition peut être exécutée à tout moment après son activation (c.à.d l'écoulement du temps est activé et non-borné).

Le profil OMEGA contient aussi un mécanisme d'observateurs qui formalise les exigences de sûreté temporisées. Un observateur est un objet special qui permet de surveiller les actions exécutées par le système et fournit un verdict concernant la (non-)satisfaction de l'exigence qu'il formalise. Il est modélisé par une classe stéréotypée «*observer*» qui a des attributs locaux et une machine à états afin de décrire l'exigence. La divergence de comportement du scénario nominal est marquée par le stéréotype «*error*» sur les états ; une exécution qui atteint un tel état viole la satisfaction de l'exigence. Afin de surveiller le système, un ensemble de sondes d'événements a été défini dont *send* qui surveille l'envoi d'un signal et *acceptsignal* qui agit de même pour traiter un signal. Le déclenchement d'une transition est défini par une clause *match* qui précise le type d'événement, le nom du signal qualifié et les attributs de l'observateur qui reçoivent des informations connexes. Sa sémantique est de se synchroniser avec l'exécution du signal mentionné dans la clause.

3.2 Un méta-modèle pour les contrats comportementaux

Pour commenter le méta-modèle de la figure 5.1, nous commençons par présenter les méta-classes qui sont réutilisées en tant que telles de *UML4SysML*. La méta-classe *Property* désigne la notion de composant dans le standard. La méta-classe *Class* indique le type des composants. Les exigences qui sont informellement utilisées en SysML sont représentées par la méta-classe *SafetyProperty* afin de pouvoir appliquer la théorie à base de contrats à des formalismes différents.

Nous avons identifié deux catégories de notions définies dans la méta-théorie précédemment décrite : (1) celles qui définissent comment modéliser un contrat et qui sont représentées dans la partie supérieure du méta-modèle et (2) celles qui définissent les relations de vérification à utiliser entre composants/contrats et qui sont représentées dans la partie inférieure de la figure 5.1.

Afin d'introduire les contrats, nous définissons tout d'abord l'hypothèse et la garantie qui sont respectivement représentées par les méta-classes *Assumption* et *Guarantee*. Les deux notions sont de type *Class*. L'intuition pour ce choix de modélisation est simple : dans la méta-théorie, une hypothèse/garantie est un type particulier de composant, il est donc naturel de le modéliser comme une classe avec un comportement décrit par une machine à états. Ainsi, nous utilisons le même langage pour les composants et les hypothèses/garanties. Cependant, dès lors que les hypothèses/garanties sont utilisées dans des relations de raffinement vérifiables dans un cadre formel, nous devons restreindre leur syntaxe par rapport aux relations dans lesquelles elles peuvent être impliquées : toutes les relations sont interdites sauf les réalisations d'interfaces nécessaires pour typer les ports.

Un contrat est représenté par la méta-classe *Contract* de type *Class* comme une structure composite contenant exactement une hypothèse et une garantie, tous les autres attributs étant interdits. De plus, un contrat n'a pas de comportement. Cette modélisation permet la réutilisation : un contrat est défini par des instances de types, tandis que les types (c.a.d hypothèse/garantie) peuvent être utilisés dans d'autres contrats.

La méta-théorie exige qu'un contrat soit un système fermé. Les actions UML étant des entrées et des sorties, cela signifie que la cible de toutes les sorties de l'hypothèse/garantie est la garantie/hypothèse et que toutes les entrées sont déclenchées par les actions réciproques. Comme dans nos systèmes la communication est réalisée par les ports et les connecteurs, cette contrainte peut être facilement exprimée sur les ports de chaque composant d'un contrat qui doivent correspondre par type et direction inverse.

De plus, le but d'un contrat est de modéliser un comportement partiel, par rapport à une seule exigence. Ce dispositif est mis en œuvre par la possibilité que la garantie modélise seulement un sous-ensemble de la signature du composant qui doit satisfaire le contrat. Nous notons la satisfaction d'un contrat par la méta-classe *Implementation* et nous expliquons dans la suite la sémantique de ce concept. Cette contrainte sur la signature d'une garantie est formalisée par rapport à l'ensemble des ports définis : le port d'une garantie doit avoir les mêmes nom, type et direction qu'un port du type du composant. La nécessité de cette contrainte est motivée par une conception dirigée par les exigences : les spécifications sont raffinées à partir d'une exigence vers des implémentations ; l'intégration de plusieurs exigences dans la même spécification entraîne une signature plus riche pour le composant.

Comme nous l'avons mentionné, la relation de satisfaction d'un contrat par un composant est représentée par la méta-classe *Implementation* au niveau des types. Cette relation de type *Dependency* est définie entre une *Classe* et un *Contract* et exprime que la classe satisfait le contrat. Un contrat peut être implémenté par plusieurs classes et une classe peut implémenter plusieurs contrats. Dans ce dernier cas, le concepteur doit spécifier pour chaque composant quel contrat doit être utilisé lors de la vérification d'une exigence. Pour cela, nous définissons une deuxième relation de type *Dependency*, nommé *ContractUse*, qui modélise la satisfaction d'un contrat au niveau des composants. Puisqu'un modèle système doit satisfaire plusieurs exigences, le concepteur doit spécifier pour chaque contrat utilisé quelle est l'exigence à satisfaire. Cette condition est modélisée à la figure 5.1 par l'association entre *ContractUse* et *SafetyProperty.* Afin d'assurer la cohérence d'un modèle, une relation *ContractUse* peut être modélisée entre une *Property* et un *Contract* si et seulement si le type du composant *implémente* le contrat.

Compte tenu de ces définitions, les relations *Implementation* peuvent être dérivées à partir des relations *ContractUse* modélisées.

La deuxième étape de la méthodologie consiste à modéliser la dominance entre un contrat plus général et un ensemble de contrats plus spécifiques. Cette relation n'est pas explicitement modélisée à la figure 5.1 car elle peut se déduire des relations *ContractUse* définies. En effet, pour que la méta-théorie soit correctement appliquée, chaque composant du système — atomique ou composé — impliqué dans la satisfaction d'une exigence doit faire partie d'une relation *ContractUse.* Le raisonnement se poursuit comme suit : à partir du contrat global et l'exigence à satisfaire, le composant qui l'implémente est obtenu. Pour chaque sous-composant, en se basant toujours sur les relations *ContractUse* qui pointent l'exigence cible, l'ensemble des contrats dominants est calculé. Afin que le calcul soit sans ambiguïté et qu'un seul ensemble de contrats dominants soit obtenu dans le contexte considéré, chaque relation *ContractUse* doit spécifier, en plus de l'exigence à satisfaire, quel est le contrat à raffiner. Ceci est représenté à la figure 5.1 par l'association *refTarget* entre *ContractUse* et *Contract.* Par conséquence, la dominance est une relation quadruple entièrement déterminée par *ContractUse*, *Contract* et *SafetyProperty.*

Le même raisonnement concernant le raffinement de la signature d'un contrat peut s'appliquer à la relation de dominance : la signature de la garantie globale peut être un sous-ensemble ou égale à l'union des signatures des garanties dominantes. Nous exprimons cette règle également sur les ports : chaque port modélisé pour la garantie globale doit être identique (par type et direction) à un port modélisé pour une des garanties dominantes.

La dernière étape du raisonnement consiste à modéliser la relation de conformité représentée par la méta-classe *Conformance* de type *Dependency* entre un *Contrat* et une *SafetyProperty.* Nous notons que plusieurs exigences peuvent être vérifiées par le même contrat.

Notre but est d'utiliser la méta-théorie décrite précédemment pour vérifier les modèles système ; dans ce cas nous devons garantir que chaque étape du raisonnement est correctement modélisée — complète et unique — par rapport à chaque exigence. Les conditions suivantes doivent être satisfaites par les modèles étendu par des contrats :

- il existe une unique relation *ContractUse* entre un *Property*, un *Contract*, un *SafetyProperty* — *reqTarget* — et un *Contract* — *refTarget* —, et
- dans un modèle, pour chaque *SafetyProperty*, il existe un contrat qui lui est conforme.

Cet ensemble de règles est formalisé en OCL [92] et détaillé en annexe A.1 afin d'assurer la correction statique d'un modèle avec des contrats et ensuite générer un ensemble d'obligations de preuve dont leur satisfaction garantit la satisfaction de l'exigence.

3.3 Instanciation du méta-modèle par un profil

L'utilisation des contrats dans un modèle UML/SysML standard implique l'instanciation du méta-modèle précédemment défini. Notre choix est d'utiliser le mécanisme des stéréotypes sur les méta-classes et de définir un profil. Ainsi, les stéréotypes applicables sur la méta-classe *Class* sont «*assumption*», «*guarantee*», «*contract*» et «*observer*». Nous rappelons que le stéréotype «*observer*» correspond à l'instantiation de la méta-classe *SafetyProperty* en OMEGA par un observateur. Pour la méta-classe *Dependency*, nous définissons les stéréotypes «c*ontractConformance*», «*contractImplementation*» et «*contractUse*», où ce dernier a deux attributs (ou valeurs étiquetées) *reqTarget* et *refTarget* qui font référence à l'exigence à satisfaire et le contrat à raffiner. Ces notions sont illustrées à la figure 5.3. De plus, afin de garantir l'instantiation correcte du méta-modèle avec des stéréotypes, nous proposons un deuxième ensemble de règles de bonne formation décrites en annexe A.2.

3.4 Travaux connexes

Les contrats en génie logiciel sont classés dans [28] en 4 catégories: *syntaxiques* qui décrivent les types qu'un composant peut gérer, *comportementaux* qui ajoutent des contraintes sur l'utilisation d'un composant, de *synchronisation* qui spécifient le comportement global et l'interaction des composants et de *qualité de service* qui peuvent quantifier le comportement attendu des composants. Nos contrats, même si nous les qualifions de comportementaux dû au fait qu'ils modélisent un comportement, font partie de la catégorie des contrats de synchronisation car ils décrivent explicitement l'ordre d'appel des requêtes et leur synchronisation. Cependant, nous estimons qu'ils peuvent également être utilisés comme syntaxiques et comportementaux : la signature du contrat précise également la signature du composant et l'hypothèse/garantie peut être considérée comme une spécification générique des pré/post-conditions pour l'utilisation du composant.

La plupart de travaux mettent l'accent sur les contrats syntaxiques et comportementaux afin de fournir un mécanisme pour le problème de composabilité. Dans [156], les auteurs font la distinction entre un contrat offert et un contrat requis par le composant. Ces contrats décrivent les types gérés par le composant durant les phases de développement. Dans [114] les contrats sont définis comme des pré/post-conditions OCL pour les opérations, qui sont ensuite validées par simulation. Dans [36, 35], le même type de contrat est utilisé pour tous les éléments de modélisation UML afin de décrire la transformation de modèles et leur sémantique d'exécution.

Le modèle de composants Kmelia [120, 11] fournit un moyen de vérifier la correction fonctionnelle des contrats de synchronisation pour les services Web, ainsi que la compatibilité entre les composants. Les contrats de synchronisation ont été considérés dans [136] entre les ports de deux composants, c.à.d définis pour un connecteur, pour vérifier leur compatibilité.

Les contrats ont également été utilisés pour les architectures synchrones modélisées en SysML et AADL dans [157, 55, 78], en proposant un raisonnement similaire à celui décrit dans la section 2.1 et dont le contrat est défini par une paire pré/post-condition en Logique Temporelle Linéaire (LTL) pour un composant. Cependant, cette théorie né définit pas une syntaxe pour les contrats en SysML ou AADL.

Pour conclure, notre théorie à base de contrats est complémentaire aux précédentes approches présentées pour SysML, car cela concerne le comportement temporisé de modèles système et la satisfaction des exigences.

4 Un modèle formel pour la sémantique de modèles SysML

Afin d'appliquer les techniques de vérification et de validation sur les modèles système, nous devons fournir un modèle formel qui décrit leur sémantique. Nous avons choisi de construire notre cadre sur une variante des automates temporisés entrée/sortie (TIOA) tel que défini dans [108], car il convient d'exprimer la sémantique des composants réactifs temporisés de SysML. En outre, il est bien défini et fournit des résultats compositionnels prêts à l'emploi qui sont demandés par la méta-théorie que nous instancions.

4.1 Une variante des automates temporisés entrée/sortie pour les modèles SysML

La sémantique d'un composant SysML est représentée par un automate temporisé entrée/sortie :

Définition 4 (Automate temporisé entrée/sortie). Un *automate temporisé* *entrée/sortie* (TIOA) $\mathcal{A}$ est une structure (X, Clk, Q, θ, I, O, V, H, D, $\mathcal{T}$) où :

- X est un ensemble fini de variables discrètes et Clk est un ensemble fini d'horloges. Nous notons par $Y = X \cup Clk$ l'ensemble des variables internes.
- $Q \subseteq val(Y)$ est un ensemble d'états dont $val(Y)$ est l'ensemble des fonctions d'évaluation pour Y. Une fonction d'évaluation est définie sur Y et elle associe à chaque variable une valeur de son domaine.
- $\theta \in Q$ est l'état initial.
- I est un ensemble d'entrées, O un ensemble de sorties et V un ensemble d'actions visibles. Nous notons par $E = I \cup O \cup V$ l'ensemble des actions externes que nous appelons dans la suite *signature de l'automate.*
- H est l'ensemble des actions internes. Nous notons par $A = E \cup H$ l'ensemble des actions exécutables.
- I, O, V et H sont des ensembles disjoints.
- $D \subseteq Q \times A \times Q$ est un ensemble de transitions discrètes.
- $\mathcal{T}$ est l'ensemble des trajectoires. Chaque trajectoire est une fonction τ : $J_\tau \to Q$, où J_τ est un intervalle réel de type $[0, d]$ ou $[0, \infty)$ avec $d \in \mathrm{R}_+$, tel que $\forall t \in J_\tau$:
  - $\tau(t)(x) = \tau(0)(x)$, $\forall x \in X$, et
  - $\tau(t)(clk) = \tau(0)(clk) + t$, $\forall clk \in Clk$.

Notons qu'il existe deux différences entre la définition précédente et celle présentée dans [108]. La première traite l'extension de TIOA avec des actions visibles, en plus des entrées, sorties et actions internes. Ces actions trouvent leur justification dans la composition entrée-sortie des composants. Lors d'une composition, l'envoi et la réception d'un signal (action) n'a besoin que de laisser une trace visible dans la sémantique SysML, alors que dans [108] il devient une sortie permettant ainsi le multicast qui est incompatible avec la sémantique SysML. La nécessité d'actions visibles est motivée par les exigences du système qui sont souvent décrites par rapport à des systèmes fermés et qui généralement mettent en œuvre la synchronisation des entrées-sorties.

La seconde différence réside dans la restriction des trajectoires à des fonctions constantes pour les variables discrètes et à des fonctions linéaires avec la dérivée égale à 1 pour les horloges. Cette restriction rend l'expressivité de notre modèle temporisée équivalente à celle des automates temporisés Alur-Dill [7], alors que la définition de [108] couvre les systèmes hybrides. Cette restriction ouvre la possibilité d'exécuter l'analyse d'accessibilité ou de vérifier des relations de simulation, qui sont indécidable pour les TIOA de [108]. Cependant, les résultats compositionnels exigés par la méta-théorie sont indépendants de cette restriction, car ils peuvent être prouvés également pour les systèmes hybrides comme décrit par [108].

Tout transition $(\mathrm{x}, a, \mathrm{x}') \in D_{\mathcal{A}}$ est noté par $\mathrm{x} \xrightarrow{a} \mathrm{x}'$. Pour une trajectoire τ, nous notons $\tau(0)$ par $\tau.\mathit{fval}$ et $\tau.\mathit{ltime}$ le supremum de son domaine. Alors $\tau.\mathit{lval} = \tau(\tau.\mathit{ltime})$. La même notation que pour les transitions est utilisée pour les trajectoires, $\mathrm{x} \xrightarrow{\tau} \mathrm{x}'$ où $\mathrm{x} = \tau.\mathit{fval}$ et $\mathrm{x}' = \tau.\mathit{lval}$. Une trajectoire τ est fermée si son domaine est un intervalle fermé.

Le comportement d'un automate temporisé entrée/sortie est donné par un ensemble d'exécutions.

Définition 5 (Exécution). Une *exécution* d'un automate temporisé entrée/sortie $\mathcal{A}$ est une séquence éventuellement infinie $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$ où

- $\forall i$, $a_i \in A$ and $\tau_i \in T$,
- $\tau_0.\mathit{fval} = \theta$,
- si τ_i n'est pas la dernière trajectoire dans α, alors τ_i est fermée et $\tau_i.\mathit{lval} \xrightarrow{a_{i+1}} \tau_{i+1}.\mathit{fval}$,
- si τ_i est la dernière trajectoire, elle peut être ouverte ou fermée, et
- si α est une séquence finie alors elle se termine par une trajectoire.

Nous notons par $\mathit{reach}(\alpha)$ l'ensemble des états atteints par l'exécution α.

Une exécution contient toutes les informations. Cependant, certains éléments ne présentent pas beaucoup d'intérêt lors du raffinement de comportement, comme l'évolution des variables internes au cours de l'écoulement du temps ou l'exécution des actions internes. Nous utilisons la notion de trace introduite dans [108] comme projection d'une exécution sur les actions externes et sur les intervalles de temps écoulés.

Définition 6 (Trace). Soit α une exécution. La *trace* de α est la restriction de α à $(E_{\mathcal{A}}, \emptyset)$, notée $\mathit{trace}(\alpha) = \alpha\lceil(E_{\mathcal{A}}, \emptyset)$, où :

- chaque $a_i \in \mathit{trace}(\alpha)$ est une action de $E_{\mathcal{A}}$, à savoir les actions de $H_{\mathcal{A}}$ sont supprimées de α, et
- chaque $\tau_i : J_{\tau_i} \to \emptyset$, $J_{\tau_i} \subseteq \mathbb{R}_+$, n'enregistre que la durée de l'écoulement du temps et ignore l'évolution des variables.

Si la trace obtenue après la projection contient des trajectoires adjacentes, l'opérateur de concaténation est appliqué afin d'obtenir une seule trajectoire. Nous notons avec $\mathit{traces}_{\mathcal{A}}$ l'ensemble de traces du TIOA $\mathcal{A}$ et par $\mathit{tracefrags}_{\mathcal{A}}(q)$ l'ensemble de fragments de traces qui débutent en q — à la place de l'état initial θ. L'ensemble des traces d'un automate peut présenter deux propriétés : fermeture sous limites et fermeture sous l'extension du temps. La fermeture sous limites modélise informellement le fait que toute séquence infinie dont les préfixes sont des traces est aussi une trace. La fermeture sous l'extension du temps indique que toute trace peut être étendue par une trajectoire ayant comme domaine un intervalle ouvert qui permet la progression du temps à l'infini sans qu'aucune autre action visible ne se produise. Les définitions formelles de ces deux notions sont celles présentées dans [108].

Nous définissons un opérateur de composition parallèle pour permettre aux automates de communiquer et de s'exécuter en parallèle. La définition suivante présente les conditions qui doivent être satisfaites pour composer deux automates.

Définition 7 (Composants compatibles). Deux automates temporisés entrée/-sortie $\mathcal{A}_1$ et $\mathcal{A}_2$ sont compatibles si $Y_{\mathcal{A}_1} \cap Y_{\mathcal{A}_2} = H_{\mathcal{A}_1} \cap A_{\mathcal{A}_2} = H_{\mathcal{A}_2} \cap A_{\mathcal{A}_1} = V_{\mathcal{A}_1} \cap A_{\mathcal{A}_2} = V_{\mathcal{A}_2} \cap A_{\mathcal{A}_1} = O_{\mathcal{A}_1} \cap O_{\mathcal{A}_1} = I_{\mathcal{A}_1} \cap I_{\mathcal{A}_2} = \emptyset$.

Syntaxiquement, l'opérateur de composition parallèle modélise la synchronisation entre une entrée et une sortie et l'entrelacement de toutes les autres actions. La communication asynchrone est réalisée en faisant la différence entre une transition discrète avec une entrée qui stocke le message dans une file d'attente prédéfinie et une transition avec une action interne qui modélise la consommation du message de la file d'attente.

Définition 8 (Composition parallèle). Si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont deux automates temporisés entrée/sortie compatibles, alors leur composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ est définie comme étant le tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ où :

- $X = X_1 \cup X_2$ and $Clk = Clk_1 \cup Clk_2$.
- $Q = \{\mathrm{x}_1 \cup \mathrm{x}_2 | \mathrm{x}_1 \in Q_1, \mathrm{x}_2 \in Q_2\}$.
- $\theta = \theta_1 \cup \theta_2$.
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$.
- $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$.
- $V = V_1 \cup V_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$.
- $H = H_1 \cup H_2$.
- D est l'ensemble des transitions discrètes donné par les règles suivantes : pour chaque $\mathrm{x} = \mathrm{x}_1 \cup \mathrm{x}_2$, $\mathrm{x}' = \mathrm{x}'_1 \cup \mathrm{x}'_2 \in Q$ et pour chaque $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ si et seulement si $\forall i \in \{1, 2\}$
    1. $a \in A_i$ and $\mathrm{x}_i \xrightarrow{a}_i \mathrm{x}'_i$ où
    2. $a \notin A_i$ and $\mathrm{x}_i = \mathrm{x}'_i$.
- $\tau \in \mathcal{T} \Leftrightarrow \tau \lceil X_i \in \mathcal{T}_i$, $i \in \{1, 2\}$.

La seule différence entre cette définition et celle présentée dans [108] est liée à la signature de l'automate temporisé entrée/sortie composite : les ensembles d'actions d'entrée et de sortie consistent en ceux qui ne sont pas appariées, tandis que les entrées-sorties appariées deviennent des actions visibles. Par différence, dans [108] les entrées-sorties appariées deviennent sorties, ce qui signifie que les sorties sont traitées comme des émissions multiples, ce qui n'est pas conforme à la sémantique SysML.

Théorème 2. $(\mathcal{A}, \|)$ *est un monoïde commutatif.*

Comme dans [108], nous utilisons l'inclusion de l'ensemble des traces comme relation de raffinement entre automates.

Définition 9 (Composants comparables). Deux automates temporisés entrée/sortie $\mathcal{A}_1$ et $\mathcal{A}_2$ sont comparables s'ils ont la même signature, c.à.d $E_{\mathcal{A}_1} = E_{\mathcal{A}_2}$.

Définition 10 (Conformité). Soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates temporisés entrée/sortie comparables. $\mathcal{A}_1$ *raffine* (*est conforme à*) $\mathcal{A}_2$, noté $\mathcal{A}_1 \preceq \mathcal{A}_2$, si $traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$.

Notons que nous utilisons la relation de raffinement entre les composants pour vérifier la conformité dans la quatrième étape de la méthodologie entre un contrat et l'exigence à satisfaire. Cela génère l'obligation de preuve suivante : $A \parallel G \preceq \varphi$. Dans ce qui suit, nous allons utiliser le terme *est conforme à* pour désigner le raffinement des composants. Les théorèmes 3 et 4 (ce dernier correspondant au théorème de composabilité 8.5 de [108]) qui sont requis par la méta-théorie peuvent être facilement étendus à notre variante de TIOA.

Théorème 3. *La relation de conformité est une relation de préordre sur l'ensemble des composants comparables.*

Théorème 4 (Composabilité). *Soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates temporisés entrée/sortie comparables tels que $\mathcal{A}_1 \preceq \mathcal{A}_2$, et E un automate temporisé entrée/sortie compatible avec $\mathcal{A}_1$ et $\mathcal{A}_2$. Alors $\mathcal{A}_1 \parallel E \preceq \mathcal{A}_2 \parallel E$.*

4.2 Transformation des modèles SysML en modèles TIOA

La transformation d'un système à base de composants modélisé avec les notions SysML mentionnées dans la section 3.1 dans un réseau d'automates temporisés entrée/sortie est relativement simple. Nos règles de transformation suivent la même stratégie qui a été décrite dans des travaux antérieurs comme [122, 109] ou récents [87].

Pour chaque composant SysML K, un automate temporisé entrée/sortie $\mathcal{A}_K$ est généré. La raison de transformer directement un composant résulte du manque de mécanismes dans le formalisme TIOA présenté à définir des types structurés et des opérations d'instanciation. L'ensemble des horloges $Clk_{\mathcal{A}_K}$ contient tous les attributs définis par le composant de type *Timer*, tandis que tous les autres attributs modélisés forment l'ensemble des variables discrètes $X_{\mathcal{A}_K}$. Il est supposé que chaque automate $\mathcal{A}_K$ contient deux variables discrètes implicites : la file d'attente *queue* qui stocke toutes les entrées et les expédie pour être traitées, et l'état de contrôle actuel *location* de la machine à états. Au niveau du type du composant, on peut modéliser des relations : les associations sont gérées en fonction de leurs extrémités qui représentent des attributs dans les classes correspondantes, tandis que la généralisation entre les classes est aplatie et tous les attributs hérités sont dupliqués dans l'automate correspondant au composant fils.

L'ensemble des états de l'automate $\mathcal{A}_K$ est donné par l'évaluation de toutes les variables, où l'état initial θ_K soit contient une valeur définie par l'utilisateur lors de son initialisation ou une valeur prédéfinie comme 0 pour les horloges ou $\emptyset$ pour la file d'attente.

Le comportement du composant est modélisé par la machine à états de son type qui décrit les transitions et les trajectoires de l'automate. Une transitions d'une machine à états est définie entre un état de contrôle source s et un état de contrôle cible s' sur laquelle nous pouvons évaluer une garde et exécuter plusieurs effets. Une transition est activée par un délai de temps ou un déclenchement d'action. Ainsi, pour chaque transition de la machine à états un ensemble de transitions TIOA est généré entre deux états q et q' où $q.location = s$ et $q'.location = s'$. La garde modélise les conditions pour lesquelles la transition existe étant donné qu'elles sont satisfaites dans l'état de départ q, sinon aucune transition n'est générée. Dans chaque état de l'automate, une transition prédéfinie pour chaque entrée a existe. Son effet est d'ajouter le signal à la file d'attente, c.à.d $q'.queue = [q.queue; a]$. Ensuite, un déclencheur m se transforme en une transition exécutant une action interne $\downarrow m$ qui consomme le message m, ainsi $s.queue = [m, a]$ et $s'.queue = a$. L'ensemble des effets définis sur une transition peut consister en plusieurs sorties et affectations. Pour chaque effet une transition TIOA indépendante est générée. L'envoi (ou *sendAction*) devient une transition avec une sortie tel que soit l'état cible est l'état de contrôle cible s'il y a seulement cet effet modélisé ou un état intermédiaire est généré si l'effet est structuré. Cette transition se synchronisera à la composition avec la transition modélisant l'entrée et modifiera la valeur de la file d'attente. L'affectation pour les variables discrètes et les horloges est transformée en une transition TIOA qui existe si et seulement si q' peut être obtenu à partir de q en appliquant la modification.

Par défaut, le temps écoulé dans chaque état de l'automate est donné par l'ensemble des trajectoires possibles définies sur $\{[0,t]|t \in \mathbb{R}_+\} \cup \{[0,\infty)\}$. Cet ensemble de trajectoires peut être contrôlé par les étiquettes d'urgence des transitions sortantes de $s = q.location$ dans la machine à états comme suit:

- *lazy* n'ajoute pas de restrictions,
- *eager* sans garde d'horloge restreint l'ensemble des trajectoires à la trajectoire point,
- *eager* avec une garde d'horloge restreint l'ensemble des trajectoires afin qu'ils finissent dans le plus petit t où la garde est évaluée à vrai.

L'ensemble des signaux qui peuvent être manipulés par l'automate sont définis par les types des ports, alors que leur direction est donnée par la direction du port. Donc, toutes les réceptions de signaux modélisées dans les interfaces fournies définissent l'ensemble des entrées que l'automate peut manipuler et toutes les réceptions de signaux modélisées dans les interfaces requises définissent l'ensemble des sorties que l'automate peut exécuter. L'ensemble des actions visibles pour un automate qui correspond à un composant atomique est l'ensemble vide, $V_{\mathcal{A}_K} = \emptyset$. Les actions internes $H_{\mathcal{A}_K}$ sont définies pour chaque transition définissant soit une garde, un déclencheur ou une affectation. Une attention particulière doit être portée aux noms des signaux car un modèle contient généralement plusieurs composants du même type et ils réagissent aux mêmes stimuli, en contradiction avec la condition de compatibilité. Notre solution est de renommer les signaux contradictoires dans les automates émetteur/récepteur en ajoutant leur nom qualifié et les ports traversés, qui peuvent être calculés statiquement via les connecteurs. Ensuite, si l'automate récepteur traite un signal ayant plusieurs expéditeurs, la transition qui traite le signal est dupliquée pour chaque expéditeur où le déclenchement contient les nouveaux noms définis du signal. Comme l'envoi multiple est explicitement modélisé dans une conception SysML par différents signaux, le seul changement à effectuer est de renommer éventuellement les signaux sortants.

Enfin, un automate composite se traduit par un automate obtenu en appliquant l'opérateur de composition parallèle sur ses composants TIOA. Un observateur se traduit par un automate temporisé entrée/sortie n'ayant que des actions visibles.

4.3 Implémentation avec IFx2

La transformation présentée précédemment est mise en œuvre dans la boîte à outils IFx2 pour les modèles OMEGA, en mettant à jour le compilateur *uml2if* compatible uniquement aux modèles UML 1.3. Deux tâches devaient être effectuées: (1) ajouter tous les éléments de modélisation UML 2.x spécifiques (par exemple les ports, les connecteurs, les structures composites) et s'assurer que les règles de typage fort sont satisfaites et (2) mettre en œuvre la transformation vers TIOA. En outre, le simulateur interactif de modèles OMEGA a été mis à jour pour prendre en compte les nouveaux éléments de modélisation.

En ce qui concerne les choix technologiques, *uml2if* est un outil propriétaire développé en Java, Eclipse UML 2.x et Eclipse EMF. Le compilateur prend en entrée un modèle au format XMI 2.0 et produit la représentation textuelle IF du réseau TIOA. Le compilateur implémente la transformation décrite précédemment pour les modèles système étendus par des contrats. La génération des obligations de preuve est actuellement en cours de développement dans IFx2. Nous mentionnons que cet outil a été évalué sur des modèles de systèmes de qualité industrielle.

5 Raisonnement formel avec contrats

Jusqu'à présent, nous avons complètement défini — syntaxe et sémantique — l'environnement à base de composants pour lequel nous voulons utiliser le raisonnement à base de contrats. Dans cette section, nous déclinons celui des contrats pour la théorie de ces composants et montrons que cette instanciation de la métathéorie décrite dans la section 2.1 peut être appliquée à des modèles système en prouvant la satisfaction des résultats de compositionnalité nécessaires.

5.1 Théorie à base de contrats pour les TIOA

Les contrats ont été introduits dans SysML dans la section 3.2 où nous avons défini leur syntaxe. Afin de les utiliser pour la vérification de la satisfaction d'une exigence, nous devons définir leur sémantique.

Définition 11 (Environnement). Un *environnement Env* pour un composant K est un automate temporisé entrée/sortie compatible avec K pour lequel les conditions suivantes sont satisfaites : $I_{Env} \subseteq O_K$ et $O_{Env} \subseteq I_K$.

Définition 12 (Composant fermé/ouvert). Un composant K est *fermé* si $I_K = O_K = \emptyset$. Un composant est *ouvert* s'il n'est pas fermé.

Les composants fermés résultent de la composition de composants ayant des interfaces complémentaires, comme c'est souvent le cas entre un composant et son environnement. Cependant, dans la définition 11 nous modélisons des environnements partiels. Ce choix est motivé par l'architecture multi-couche des systèmes.

Définition 13 (Contrat). Un *contrat* pour un composant K est une paire (A, G) d'automates temporisés entrée/sortie tels que:

- leur composition donne un système fermé, c.a.d $I_A = O_G$ et $I_G = O_A$, et
- la signature du composant G est un sous-ensemble de la signature du composant K, c.a.d $I_G \subseteq I_K$, $O_G \subseteq O_K$ et $V_G \subseteq V_K$.

On note par la signature d'un contrat la signature de sa garantie.

La satisfaction d'un contrat a été introduite dans la définition 2 en se basant sur la relation de raffinement dans un contexte. Dans notre approche, le raffinement dans un contexte repose à son tour sur la relation de conformité (voir définition 10). Puisque nous autorisons qu'un composant concret K_i puisse avoir une signature plus grande que l'abstrait K_j et que la conformité peut être définie seulement entre composants comparables, nous devons composer chacun des membres de la relation de conformité obtenue avec des automates temporisés entrée/sortie supplémentaires de sorte que la condition de compatibilité soit vérifiée.

Définition 14 (Raffinement dans un contexte). Soient K_1 and K_2 deux composants tels que $I_{K_2} \subseteq I_{K_1} \cup V_{K_1}$, $O_{K_2} \subseteq O_{K_1} \cup V_{K_1}$ et $V_{K_2} \subseteq V_{K_1}$. Soit *Env* un environnement pour K_1 compatible avec K_1 et K_2. K_1 *raffine* K_2 *dans le contexte* *Env*, noté $K_1 \sqsubseteq_{Env} K_2$, si

$$K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K' \parallel Env'$$

où K' et Env' sont définis tels que les deux membres de la relation de conformité sont fermés et comparables :

- $K' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \backslash I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \backslash O_{K_2}) \cup (V_{K_1} \cap I_{K_2})), (V_{K_1} \backslash E_{K_2}), \emptyset, \mathcal{D}_{K'}, 2^{[\mathbb{R}_+^0]})$ où ϕ est la fonction $\emptyset \rightarrow \emptyset$, $\mathcal{D}_{K'} = \{(\phi, a, \phi) | \forall a \in E_{K'}\}$ et $2^{[\mathbb{R}_+^0]} = \{[0, t] | t \in \mathbb{R}_+\} \cup \{[0, \infty)\}$.
- $Env' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \setminus I_{Env}), (I_{K_1} \setminus O_{Env}), \emptyset, \emptyset, \mathcal{D}_{Env'}, 2^{[\mathbb{R}_+^0]})$ où ϕ est la fonction $\emptyset \rightarrow \emptyset$ et $\mathcal{D}_{Env'} = \{(\phi, a, \phi) | \forall a \in E_{Env'}\}$.

Informellement, Env' est un environnement partiel ayant comme signature les actions de K_1 qui ne sont pas présentes parmi les actions de *Env* de telle sorte que $K_1 \parallel Env \parallel Env'$ soit un composant fermé. K' est un composant qui réagit aux actions définies comme la différence entre les signatures de K_1 et K_2 tel que

K_1 et $K_2 \parallel K'$ sont comparables. En outre, ces définitions satisfont les conditions suivantes : $K_1 \parallel Env \parallel Env'$ et $K_2 \parallel K' \parallel Env \parallel Env'$ sont fermés et comparables. Leur comportement est le plus simple : toutes les actions sont actives à tout moment et le temps peut s'écouler à l'infini.

Les relations d'inclusion particulières entre les signatures de K_1 et K_2 dans la définition sont dues au fait que K_1 et K_2 peuvent être des composants composés : $K_1 = K_1' \parallel K_3$ et $K_2 = K_2' \parallel K_3$, où $I_{K_2'} \subseteq I_{K_1'}$, $O_{K_2'} \subseteq O_{K_1'}$ et $V_{K_2'} \subseteq V_{K_1'}$. Cela se produit en particulier lorsque K_2' est une garantie pour K_1'. Puis, par la composition, les actions de K_3 peuvent correspondre aux actions de K_1' sans avoir de correspondants dans K_2'. Ce cas impose également le terme $V_{K_1} \cap O_{K_2}$ pour les entrées de K', car les sorties supplémentaires de K_2 peuvent correspondre à un autre composant, et le terme $V_{K_1} \cap I_{K_2}$ pour les sorties de K'.

Cette définition du raffinement dans un contexte satisfait les conditions décrites dans la section 2.1 et requises par la méta-théorie comme suit.

Théorème 5. *Étant donnés un ensemble $\mathcal{K}$ de composants comparables et un environnement Env pour les composants appartenant à $\mathcal{K}$, le raffinement dans le contexte Env est une relation de préordre.*

Proposition 1. *Soient K_1, K_2, K_3 trois composants* pas nécessairement comparables *et Env un environnement tel que $K_1 \sqsubseteq_{Env} K_2$ et $K_2 \sqsubseteq_{Env} K_3$. Alors $K_1 \sqsubseteq_{Env} K_3$.*

Théorème 6 (Compositionnalité). *Soient K_1 et K_2 deux composants et Env un environnement compatible avec K_1 et K_2 tel que $Env = Env_1 \parallel Env_2$. Alors $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1$.*

Théorème 7 (Raisonnement circulaire). *Soient K un composant, Env son environnement et $\mathcal{C} = (A, G)$ un contrat pour K tel que K et G sont compatibles avec Env et A. Si*

1. *$traces_G$ est fermé sous limites,*
2. *$traces_G$ est fermé sous l'extension du temps,*
3. *$K \sqsubseteq_A G$ et*
4. *$Env \sqsubseteq_G A$*

alors $K \sqsubseteq_{Env} G$.

La correction du raisonnement circulaire est le résultat principal qui garantit la correction du raisonnement à base de contrats que nous utilisons.

La deuxième étape du raisonnement à base de contrats consiste à prouver la relation de raffinement entre contrats afin d'écarter les composants à partir de cette étape. Comme notre théorie satisfait les résultats de compositionnalité requis par la méta-théorie, nous pouvons utiliser la condition suffisante pour la dominance écrite ci-dessous en utilisant notre notation. La dominance consiste alors à vérifier plusieurs relations de satisfaction sur des automates temporisés entrée/sortie abstraits qui sont plus faciles à manipuler.

Théorème 8. $\{C_i\}_{i=1}^n$ *domine* C *si, pour tout* i*,* $traces_{G_i}$ *et* $traces_G$ *sont fermés sous limites et sous l'extension du temps et*

$$\begin{cases} G_1 \parallel ... \parallel G_n \sqsubseteq_A G \\ A \parallel G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n \sqsubseteq_{G_i} A_i, \ \forall i \end{cases}$$

5.2 Expressivité des contrats

Cette théorie peut s'appliquer à des modèles systèmes étendus par des contrats si le composant jouant le rôle de la garantie satisfait deux conditions importantes : fermeture sous limite et fermeture sous l'extension du temps. Nous discutons ici des restrictions qui sont imposées par ces contraintes sur le langage de modélisation des contrats.

La fermeture sous limites modélise que toute trace peut être prolongée indéfiniment par des fragments de traces, faisant en sorte que le résultat soit aussi une trace. Cette contrainte est garantie par défaut par une catégorie d'automates temporisés entrée/sortie : lemme 4.20 de [108] prouve qu'un automate avec non-déterminisme interne fini a son ensemble de traces fermé sous limites.

Définition 15 (Non-déterminisme interne fini). Un automate présente du *non-déterminisme interne fini* si :

1. L'ensemble de ses états initiaux θ est fini, et
2. $\forall \mathrm{x} \in Q$, $\forall \beta \in \mathit{tracefrags}(\mathrm{x})$, l'ensemble $\{\alpha.\mathit{lval} | \alpha \in \mathit{frags}(\mathrm{x}), \alpha$ *a une trajectoire finale finie et* $\mathit{trace}(\alpha) = \beta\}$ est fini.

La première condition est satisfaite par défaut par nos automates temporisés entrée/sortie qui définissent un seul état initial correspondant à l'état initial de la machine à états qu'ils représentent. La seconde condition implique que, pour toute trace, il n'existe pas un ensemble infini de fragments d'exécution. Pour que cette condition soit satisfaite, des cycles de transitions silencieuses ne peuvent pas être modélisés dans la machine à états. Par transition silencieuse nous entendons une transition définie pour une action interne, sauf le traitement du signal. Un cycle de transitions silencieuses peut modifier les valeurs des variables conduisant ainsi à un ensemble infini d'états finaux. Deuxièmement, la sémantique temporisée pour toutes les transitions internes est *eager*. En effet, une sémantique *lazy* génère un non-déterminisme infini, car la transition peut être exécutée à tout moment pendant l'écoulement du temps et ainsi changer infiniment l'état du système. La définition d'une transition avec des actions internes comme *eager* et interdisant la modélisation des cycles de transitions silencieuses assure que l'automate est non-Zeno, c.a.d n'exécute pas une infinité d'actions internes en un temps fini. Ces conditions sont suffisantes pour assurer le non-déterminisme interne fini pour un automate.

La fermeture sous l'extension du temps permet l'écoulement du temps en tout état de l'automate. Un moyen facile de modéliser le progrès du temps dans tous les états d'une machine à états est de stéréotyper les transitions sortantes comme *lazy*. Toutefois, ce n'est pas nécessaire pour tous les types de transitions, et il est effectivement suffisant que les transitions qui exécutent des sorties soient *lazy*. Cela permet aux garanties de spécifier des contraintes de progrès de temps plus précises, par exemple en spécifiant les transitions consistant uniquement en actions internes (c.à.d calculs internes, mais aussi l'action interne de consommation de requête $\downarrow$) comme *eager*. En effet, ces paramètres d'urgence sont suffisants pour assurer la fermeture sous l'extension du temps, étant donné que les machines à états sont non-Zeno.

Nous remarquons que cette restriction ne nous permet pas de préciser dans une garantie une limite supérieure quand une sortie/action visible doit se produire, mais seulement des conditions plus faibles de type $t \geq deadline$, où t est une horloge. C'est une limitation de l'expressivité des garanties dans notre théorie.

5.3 Vérification automatique des obligations de preuve

La théorie à base de contrats que nous avons définie est basée sur la relation d'inclusion des traces. Cependant cette relation est indécidable dans le cas général et ne peut pas être automatiquement vérifiée par des outils à l'exception de certaines catégories d'automates temporisés [131, 155]. Dans ce qui suit, nous décrivons notre technique pour la vérification automatique basée sur l'analyse d'accessibilité d'automates de propriété, qui s'appuie sur le fait que le composant abstrait représente une propriété de sûreté déterministe. Cette méthode est mise en œuvre dans la boîte à outils IFx [34] qui permet de vérifier et simuler des automates temporisés communicants asynchrones.

Notre méthode utilise la notion d'automate de propriété. Un automate de propriété est la définition complète d'une exigence de sûreté : il définit un état d'erreur π vers lequel les comportements incorrects mèneront et se synchronise avec le composant étudié $\mathcal{C}$ sur les actions communes. Le raisonnement de la satisfaction d'un contrat est le suivant : (1) transformer la garantie en un automate de propriété, (2) exécuter en parallèle le composant $\mathcal{C}$ et l'automate de propriété et (3) explorer l'espace d'états pour vérifier si l'état d'erreur π peut être atteint. Atteindre l'état d'erreur π signifie la violation de la satisfaction d'un contrat.

Nous commençons par définir le processus de transformation d'une propriété de sûreté déterministe en un automate de propriété. Le mécanisme est similaire à celui défini dans [41] et plus tard utilisé pour automatiser le raisonnement hypothèse-garantie dans l'outil LTSA [84, 30].

Définition 16 (Automate temporisé de propriété). Soit un automate temporisé entrée/sortie *déterministe* $\mathcal{A} = (X_{\mathcal{A}}, Clk_{\mathcal{A}}, Q_{\mathcal{A}}, \theta_{\mathcal{A}}, I_{\mathcal{A}}, O_{\mathcal{A}}, V_{\mathcal{A}}, H_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$. L'*automate temporisé de propriété* correspondant à $\mathcal{A}$ est défini par l'automate temporisé entrée/sortie suivant $\mathcal{O}_{\mathcal{A}} = (X_{\mathcal{A}}, Clk_{\mathcal{A}}, Q, \theta_{\mathcal{A}}, \emptyset, \emptyset, V, H_{\mathcal{A}}, \mathcal{D}, \mathcal{T}_{\mathcal{A}})$ où:

- $Q = Q_{\mathcal{A}} \cup \{\pi\}$, où π est un état d'erreur additionnel,
- $V = I_{\mathcal{A}} \cup O_{\mathcal{A}} \cup V_{\mathcal{A}}$,
- $\mathcal{D} = \mathcal{D}_{\mathcal{A}} \cup \{(\mathrm{x}, a, \pi) | \mathrm{x} \in Q_{\mathcal{A}}, a \in V$ tel que $(\nexists\ \mathrm{x}'.(\mathrm{x}, a, \mathrm{x}') \in \mathcal{D}_{\mathcal{A}}) \wedge (\nexists\ \varepsilon \in H_{\mathcal{A}} \wedge \mathrm{x}' \in Q_{\mathcal{A}}.(\mathrm{x}, \varepsilon, \mathrm{x}') \in \mathcal{D}_{\mathcal{A}})\}$.

L'idée de cette transformation est que les séquences d'actions qui ne sont pas explicitement modélisées devraient être considérées comme des comportements erronés. Comme un automate temporisé de propriété est utilisé pour surveiller un composant fermé, nous considérons que sa signature ne contient que des actions visibles, correspondant aux entrées, sorties et actions visibles de $\mathcal{A}$. Puis, dans chaque état de l'automate à partir duquel il n'y a pas de transition interne sortante, nous complétons l'ensemble des transitions avec celles manquantes : pour chaque action visible il doit y avoir une transition discrète soit conduisant à un état défini en $\mathcal{A}$ ou à π. Ainsi, les actions menant à l'état d'erreur π ne sont pas autorisées à se produire dans une séquence temporelle décrite par $\mathcal{A}$.

Nous remarquons que la définition de l'automate temporisé de propriété est similaire à la modélisation des exigences avec des observateurs dans OMEGA. En effet, un observateur formalise une propriété de sûreté en modélisant l'état d'erreur π via le stéréotype d'erreur et ne définit que l'ensemble d'actions visibles, c.à.d aucune entrée ou sortie. Par conséquent, le même mécanisme de vérification peut être appliqué pour vérifier la relation de conformité. Dans ce cas, π n'est pas ajouté à $\mathcal{O}_\varphi$ s'il a été modélisé par l'utilisateur.

Cependant, pour que cette méthode fonctionne, le composant $\mathcal{A}$ doit être une propriété déterministe de sûreté à la fois pour les actions visibles et les actions internes. Pour les actions internes, le déterminisme signifie qu'il y a au plus une transition sortant d'un état. La garantie présente déjà du non-déterminisme interne fini. Par conséquent, afin d'obtenir le déterminisme, nous limitons la deuxième condition de la définition 15 de telle sorte que le cardinal de l'ensemble d'états finaux soit égal à 2 : l'état final peut être soit l'état initial si aucune transition interne n'a été tirée ou l'état obtenu par l'exécution de la transition. Nous remarquons que ces conditions doivent tenir dans le cadre des automates temporisés entrée/sortie. Cela implique que, dans une machine à états SysML, on peut modéliser plusieurs transitions internes sortantes si elles ne sont pas actives en même temps, par exemple deux transitions ayant des gardes disjointes.

La synchronisation à l'exécution entre $\mathcal{C}$ et l'automate de la propriété $\mathcal{O}_\mathcal{A}$ est définie par l'opérateur de composition suivant, noté $\bowtie$. Il est similaire à l'opérateur de composition parallèle précédemment décrit à la définition 8 avec synchronisation sur les actions visibles communes et entrelacement des autres actions. L'opérateur peut s'appliquer sur deux automates temporisés entrée/sortie s'ils ne partagent pas d'actions internes et s'ils ne définissent pas d'entrées/sorties. Cette dernière condition est motivée par le fait que l'automate de propriété surveille généralement un composant fermé.

Définition 17 (Composition synchrone). Soient $\mathcal{A}_1$ un composant fermé et $\mathcal{A}_2$ un automate temporisé de propriété tels que $E_{\mathcal{A}_1} \subseteq E_{\mathcal{A}_2}$. Alors $\mathcal{A}_1 \bowtie \mathcal{A}_2 = \mathcal{A}_1 \parallel \mathcal{A}_2$ où la condition de compatibilité est donnée par la contrainte $H_{\mathcal{A}_1} \cap H_{\mathcal{A}_2} = \emptyset$.

Le résultat suivant portant sur l'analyse d'accessibilité conclut le raisonnement. Le fait de ne pas atteindre l'état d'erreur au cours de toutes les exécutions du composant précédemment composé est suffisant pour satisfaire l'inclusion de traces.

Théorème 9. *Si K_2 est un automate temporisé de propriété déterministe et $reach((K_1 \parallel Env \parallel Env') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} = \emptyset$ alors $K_1 \sqsubseteq_{Env} K_2$.*

Le fait que l'automate temporisé entrée/sortie jouant le rôle de la garantie doit être déterministe est une limitation de cette méthode de vérification qui doit être prise en compte dans la méthodologie. La limitation n'est généralement pas problématique pour vérifier la satisfaction d'un contrat car les garanties doivent être modélisées comme des automates temporisés entrée/sortie fermés sous limites et sous l'extension du temps et ils peuvent souvent être rendus déterministes. Toutefois, afin d'établir la dominance, il faut aussi vérifier la satisfaction du contrat inverse, qui est plus problématique puisque nous n'exigeons pas la modélisation des hypothèses comme des propriétés de sûreté. Dans ce cas il y a trois solutions disponibles : (1) modéliser l'hypothèse comme une propriété de sûreté déterministe et appliquer cette méthode, (2) utiliser la simulation temporisée à la place de l'inclusion de traces ou (3) utiliser, si possible, l'environnement concret comme hypothèse, de sorte que la satisfaction d'un contrat inverse devienne triviale.

5.4 Diagnostic avec les contrats

Dans les obligations de preuve générées, une ou même plusieurs preuves peuvent ne pas être satisfaites. Dans ce cas, nous devons effectuer un diagnostic afin d'établir si l'exigence n'est pas satisfaite ou si l'ensemble des contrats doit être raffiné afin de prouver la satisfaction de l'exigence. Nous basons ce diagnostic sur la génération d'un contre-exemple, qui pour la méthode précédente conduira à l'état d'erreur π, et l'utilisation de l'approche CEGAR [50].

Nous pouvons distinguer deux cas pour lesquels la solution dépend de la manière d'appliquer le raisonnement, pour la conception ou pour la vérification : (1) la satisfaction d'un contrat ou un pas de dominance n'est pas satisfait ou (2) la vérification de la conformité échoue. Pour le premier cas, si nous sommes dans une approche de conception et si toutes les étapes précédentes ont été prouvées, nous devons raffiner les composants/contrats sources tels que le contre-exemple soit éliminé. Ceci garantit que les composants développés sont corrects par construction à l'égard de l'exigence. Si nous sommes dans une approche de vérification avec un système complètement modélisé, il faut raffiner le(s) contrat(s) cible car il est plus fréquent que l'abstraction soit erronée.

Pour le dernier cas, nous devons d'abord vérifier sur le système si le contre-exemple généré est incorrect à cause des abstractions définies ou valide, ce qui signifie que le système ne satisfait pas l'exigence. Pour un contre-exemple faux, il faut raffiner le contrat cible et vérifier au moins l'étape de dominance et la satisfaction du contrat inverse. Des itérations de raffinement de contrats doivent être exécutées jusqu'à ce que toutes les vérifications soient validées. Pour un contre-exemple valide, l'utilisateur doit redéfinir les implémentations, à savoir les composants atomiques pour lesquels la preuve a échoué. Pour cela, le raisonnement à base de contrats peut être appliqué dans une approche de conception afin d'obtenir les contrats corrects qui satisfont les obligations de preuve et les raffiner vers des implémentations correctes.

Nous remarquons que, dans le cas de nos outils, le contre-exemple généré est exprimé au niveau des automates temporisés entrée/sortie, tandis que le raffinement de contrats/composants a lieu dans un langage de modélisation de haut niveau. Le pont entre les deux formalismes est unidirectionnel : la transformation présentée ci-dessus est donnée à partir de SysML vers des automates temporisés entrée/sortie. Afin d'exploiter le scénario d'erreur, le concepteur doit appréhender en détail le système et faire usage de son expérience pour effectuer ce raffinement. Ce point est une question ouverte pour laquelle la recherche actuelle prévoit certaines options [56, 3] ; il constitue une de nos perspectives de travail.

5.5 Travaux connexes

Comme décrit à la section 2.2, il existe plusieurs frameworks à base de contrats disponibles pour les systèmes temporisés qui sont basés sur le raisonnement hypothèse-garantie. La méta-théorie de [142] est la seule qui propose un raisonnement circulaire permettant de réduire la dominance à un ensemble de preuves de satisfaction de contrat. A notre connaissance, cette étude est la première instanciation de la méta-théorie définie dans [142] pour les systèmes temporisés avec communication asynchrone.

Le framework à base de contrats proposé dans [63] pour les automates temporisés entrée/sortie présenté dans [61, 62] définit la satisfaction d'un contrat comme la simulation alternée temporisée sur le quotient. Formellement, la relation s'écrit $K \leq (A \parallel G)\backslash\backslash A$, où $\leq$ désigne la simulation et $\backslash\backslash$ l'opérateur de quotient. Cette théorie ne prend pas en compte le raffinement de signature entre les spécifications. En outre, l'opérateur de quotient est partiel : les conditions dans lesquelles l'opérateur peut être appliqué et ce qu'il advient si le résultat ne peut pas être calculé ne sont pas abordés. Comme cette théorie est une instanciation de la méta-théorie décrite en [17], la dominance ne peut être établie que par la composition des contrats et la vérification du raffinement entre la composition et le contrat abstrait. Enfin, ce cadre ne précise pas clairement quel type d'exigence peut être vérifiée ni comment une exigence doit être modélisée. En conséquence, l'étape de la conformité n'est pas formalisée.

Dans [44], une théorie de spécifications est développée pour les automates temporisés entrée/sortie de [7] qui sont étendus par la notion de coinvariant sur les états afin d'exprimer des hypothèses de vivacité. Ce cadre est adapté afin de vérifier des propriétés de sûreté et vivacité borné dans le temps. La relation de raffinement est identique à la nôtre, c.à.d l'inclusion des traces temporisées, mais le raffinement de signature n'est pas considéré ici alors qu'il est explicitement traité dans notre cadre. Une deuxième différence consiste en la définition du contrat : le cadre de [44] définit une théorie d'interfaces où une spécification englobe à la fois l'hypothèse et la garantie. L'avantage d'avoir des hypothèses et des garanties disjointes est discuté à la section 2.2.

La théorie de [43, 39, 42] couvre ces différences en proposant un framework de raisonnement hypothèse-garantie mais seulement pour les systèmes non-temporisés. Par conséquent, un contrat est donné par deux ensembles de traces — un pour l'hypothèse et l'autre pour la garantie, alors qu'une relation d'inclusion de traces covariante pour les entrées et contravariante pour les sorties est considérée pour la satisfaction d'un contrat. Comme pour la méta-théorie de [17], vérifier la dominance exige de composer les contrats et ensuite de vérifier le raffinement. En revanche, notre condition suffisante pour la dominance permet d'effectuer une vérification sur des composants plus abstraits. De plus en cas de violation, le contrat erroné peut être plus facilement identifié.

6 Une étude de cas issue de l'industrie : le *Solar Generation System* de l'ATV

Cette section présente le système *Solar Wing Management* (SGS) de l'*Automated Transfer Vehicle* (ATV) et sa vérification selon la technique de raisonnement à base de contrats. L'ATV, développé par Airbus Defence and Space[1] (ADS) est un cargo spatial mis en orbite par le lanceur européen Ariane 5 ayant comme objectif le ravitaillement de la Station Spatiale Internationale. Le système SGS décrit ici est responsable de la gestion des panneaux solaires qui fournissent au véhicule l'énergie nécessaire pour remplir sa mission. Il contient les chaînes fonctionnelles qui réalisent le déploiement et la rotation des panneaux solaires.

[1] http://www.astrium.eads.net/

6.1 Spécification du système

Le modèle représenté à la figure 9.1 a été obtenu par rétro-ingénierie du système actuel par les ingénieurs d'ADS. Il est décrit en SysML avec l'outil IBM Rhapsody. Le modèle possède une architecture en 4 couches structurée en un ensemble d'entités matérielles et logicielles qui captent son comportement temporisé. La figure 9.1 présente une vue de haut niveau des principaux composants, sans les détails de leur structure :

- le composant de mission et de gestion de véhicule (*MVM*) qui simule un scénario de déploiement des panneaux solaires.
- le composant *SOFTWARE* qui se compose de trois sous-composants, chacun ayant une fonction spécifique. Ils réagissent aux demandes provenant du *MVM* et contrôlent le matériel en exécutant des procédures automatisées en réponse aux demandes reçues.
- le composant *HARDWARE* qui contient les quatre panneaux solaires de l'ATV. Ce composant a plus de 70 pièces d'équipement avec plusieurs niveaux de redondance pour atteindre la fiabilité et la disponibilité en cas de défaillance. Chaque aile est maintenue dans sa position initiale par quatre systèmes de retenue et libération (HDRS). Pour que le déploiement se produise, chaque HDRS doit être coupé. Ceci est réalisé par huit couteaux thermiques (TK) pour chaque aile, deux pour chaque HDRS, un pour le cas nominal et un redondant en cas d'anomalie. Chaque fois qu'un HDRS est coupé, le mécanisme de verrouillage de l'aile évolue vers l'état déployé pour cette aile.
- le composant des unités de commande (*CU*) qui peut également être soumis à l'échec. Il a de nombreuses interconnexions nominales et redondantes avec les ailes, connexions qui sont abstraites à 4 dans la figure 9.1. Il contient 4 unités d'énergie (PCDU) et 4 unités de contrôle thermique (TCU) qui sont responsables de l'activation/désactivation des couteaux thermiques, chacun d'eux étant relié à deux ailes différentes. Deux unités de commande et de surveillance (CMU) supervisent l'ensemble du système, c.à.d toutes les demandes du logiciel transitent les CMU.

Le composant SGS décrit deux modes de fonctionnement : (1) le déploiement des panneaux solaires et (2) leur rotation. Nous ne nous intéressons ici qu'au premier mode. Au départ, les quatre panneaux solaires de l'ATV sont rangés. Leur déploiement commence par enlever les barrières de sécurité des unités de contrôle thermique. Les barrières de sécurité empêchent un déploiement non désiré des ailes en bloquant l'activation des couteaux thermiques. Puis, les HDRS sont coupés par au moins quatre des huit couteaux thermiques de chaque aile. Pour qu'un HDRS soit coupé, le couteau doit être actif pendant 50 secondes consécutives. Le déploiement de l'aile commence immédiatement après que le dernier HDRS soit coupé. Une fois le déploiement terminé, les barrières de sécurité sont restaurées.

La redondance du système, si une anomalie se produit lors de l'exécution, est explicitement modélisée pour les TK et HDRS de chaque aile, TCU et PCDU. Il y a 56 échecs possibles et chacun peut se produire à un moment arbitraire lors de l'exécution. L'hypothèse est que le système peut faire l'objet d'au plus un échec. Afin de faciliter la génération des configurations de vérification, un composant spécial *SIMULATION* est ajouté au modèle afin de commander de façon non-déterministe la défaillance d'un équipement basée sur un paramètre qui peut être fourni avant la session de vérification.

En termes de mesures, le modèle définit un total de 21 types de bloc (dont 7 sont raffinés par le biais de 24 diagrammes internes) avec 348 types de port et 372 type de connecteurs pour la communication. Au moment de l'exécution, le système comptabilise 96 instances de blocs fonctionnant en parallèle avec un total de 651 ports et 504 connecteurs.

Nous sommes intéressés à prouver que le système est tolérant à une panne. Cela signifie que pour une occurrence d'erreur, le logiciel est en mesure d'atteindre son objectif et donc de garantir que le déploiement des ailes s'effectue. Ce comportement se traduit par l'exigence suivante modélisée à la figure 9.2.

Exigence. *A la fin de la séquence de déploiement, les quatre ailes sont déployées.*

Nous formalisons cette exigence par un observateur. Nous ajoutons au modèle système un bloc *Property* dont la machine à états décrit la propriété de sûreté à vérifier/ Initialement nous attendons dans l'état *SYSTEM_IS_ON* pour l'interrogation de l'état de l'aile. Après la séquence de déploiement, un composant du logiciel vérifie l'état de verrouillage des ailes. Lorsqu'on les sollicite, les ailes cibles indiquent leur statut par un message *SGS_DEPLOY_WING_STATUS*. Lorsque l'action est exécutée, l'automate passe dans l'état *VERIFY_DEPLOYMENT* où il vérifie le statut transmis. Si c'est *LOCKED_DEPLOYED*, alors il attend un autre événement de l'interrogation pour une autre ou la même aile. Sinon, quelque chose d'imprévu s'est produit lors du déploiement et l'observateur avance à l'état d'erreur *NO_DEPLOYMENT*. Atteindre l'état d'erreur lors de la vérification signifie que l'exigence n'est pas satisfaite.

6.2 Application de la théorie à base de contrats

Nous commençons par identifier les composants qui représentent le système à étudier S et l'environnement E. Comme l'exigence est exprimée par rapport au comportement des quatre ailes du composant *HARDWARE*, nous considérons que le système S est défini par le composant *HARDWARE* et les K_i par les composants *WINGi*, $i = \overline{1,4}$. L'environnement du système est donné par les composants avec lesquels il communique : la communication bidirectionnelle est établie entre *CU* et *HARDWARE*, tandis que *CU* dépend du comportement de *SOFTWARE* et *MVM*. Ainsi, l'environnement E de la figure 3.1 est représenté ici par la composition de *MVM*, *SOFTWARE* et *CU*.

La première étape de la méthodologie consiste à définir un contrat $\mathcal{C}_i = (A_i, G_i)$ pour chaque *WINGi* et à prouver que *WINGi* satisfait $\mathcal{C}_i$, $i = \overline{1,4}$. Nous avons choisi pour *WINGi* d'utiliser comme hypothèse l'environnement concret du système *HARDWARE* composé d'une abstraction *WAj* pour chaque *WINGj* avec $j \neq i$. Nous proposons l'abstraction suivante *WAj* : l'aile consomme toutes les demandes provenant de l'environnement et répond à toute demande d'état avec *déployé*. Puis l'hypothèse A_i est donnée par la composition parallèle de *MVM*, *SOFTWARE*, *CU* et *WAj* avec $j \neq i$. Cette abstraction de l'environnement est suffisante pour réduire considérablement l'espace d'états du système, sachant que l'explosion exponentielle dans le modèle original est principalement due au parallélisme des pièces du matériel qui sont abstraites aux trois composants atomiques *WAj*. Nous voulons garantir que même si *WINGi* présente une panne, il finit par être déployé.

Contrat $\mathcal{C}_i = (A_i, G_i)$ où

- $A_i = MVM \parallel SOFTWARE \parallel CU \parallel (\parallel_{j \neq i} WAj)$.
- $G_i = WAi$: l'aile répond aux demandes concernant son statut avec *déployé* et ignore toutes les autres demandes

Le contrat est modélisé à la figure 9.5, tandis que la figure 9.6 présente le comportement de la garantie. Puisque nous utilisons comme hypothèse l'environnement concret, la signature de la garantie est celle du composant. Pour cette raison, nous devons ajouter des transitions de consommation dans chaque état et pour toutes les entrées correspondant au processus de déploiement de l'aile. En outre, nous remarquons que le même type est utilisé pour modéliser dans les 4 contrats les garanties mais aussi les abstractions des ailes ; cela montre la propriété de réutilisation de notre cadre de modélisation à base de contrats. Nous préférons ne pas introduire une nouvelle notation et utiliser *WAi* pour désigner également les garanties. En outre, on peut remarquer que la garantie est plus forte que la projection de l'exigence sur *WINGi* : l'abstraction *WAj* peut être également soumise à une panne puisque ce cas n'est pas exclu de son comportement ; donc, la propriété de la tolérance aux pannes que nous vérifions via les contrats est plus forte que celle qui était prévue. Nous garantissons que le système est tolérant aux 4 pannes, si les pannes se produisent dans des ailes séparées.

La deuxième étape de la méthode consiste à définir un contrat global $\mathcal{C} = (A, G)$ pour *HARDWARE* et prouver que le contrat est dominé par $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$. Encore une fois, nous utilisons comme hypothèse A l'environnement concret du *HARDWARE*. La garantie G est la composition des quatre *WAi*. Tous les *WAi*, $i = \overline{1, 4}$, et G remplissent les conditions de fermeture nécessaires d'application du théorème 8.

Contrat $\mathcal{C} = (A, G)$ où

- $A = \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU}$
- G : pour chaque aile, la garantie répond aux demandes de statut avec *déployé*, tandis que toutes les autres demandes sont ignorées.

La troisième étape du raisonnement consiste à prouver la satisfaction du contrat inverse $\mathcal{C}^{-1}$. Cette vérification est triviale puisque l'environnement concret est utilisé comme hypothèse et l'obligation de preuve s'écrit : $\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \sqsubseteq_G \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU}$.

La dernière étape consiste à vérifier que la composition $A \parallel G$ est conforme à l'exigence.

Les preuves de ces quatre étapes ont été automatiquement vérifiées avec la boîte à outils OMEGA-IFx et la méthode décrite dans la section 5.3. Pour chaque étape de la méthodologie nous avons modélisé manuellement les contrats : les hypothèses en tant que blocs ont été connectées via les ports aux autres composants et les garanties ont été considérées comme des composants indépendants. La première étape a donné lieu à quatre configurations possibles avec une aile concrète et trois abstraites qui ont été vérifiées chacune à l'égard des 14 pannes possibles. Le temps moyen en secondes nécessaire à la vérification de la relation de satisfaction pour chaque contrat et par rapport à chaque catégorie de panne est présenté à la table 9.1. Même si le modèle système semble symétrique, les unités de commande n'ont pas un comportement similaire et, en raison de leurs interconnexions avec les ailes, l'espace d'états du système pour *WING1* et *WING3* est plus grand que celui de *WING2* et *WING4* : le composant *CMU1* est responsable de *WING1* et *WING3* pendant le déploiement de l'aile, mais transfère les requêtes vers les quatre ailes pendant la préparation, alors que *CMU2* ne gère que le déploiement de *WING2* et *WING4* Pour la seconde étape, les obligations de preuve suivantes doivent être vérifiées :

1. $\mathit{WA1} \parallel \mathit{WA2} \parallel \mathit{WA3} \parallel \mathit{WA4} \sqsubseteq_{\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU}} G$
2. $\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA2} \parallel \mathit{WA3} \parallel \mathit{WA4} \sqsubseteq_{WA1} \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA2} \parallel \mathit{WA3} \parallel \mathit{WA4}$
3. $\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA3} \parallel \mathit{WA4} \sqsubseteq_{WA2} \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA3} \parallel \mathit{WA4}$
4. $\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA2} \parallel \mathit{WA4} \sqsubseteq_{WA3} \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA2} \parallel \mathit{WA4}$
5. $\mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA2} \parallel \mathit{WA3} \sqsubseteq_{WA4} \mathit{MVM} \parallel \mathit{SOFTWARE} \parallel \mathit{CU} \parallel \mathit{WA1} \parallel \mathit{WA2} \parallel \mathit{WA3}$

Nous remarquons que les quatre dernières preuves sont triviales puisque le même membre se trouve des deux côtés des relations. Seule la vérification du raffinement du contrat global est nécessaire, ce qui a duré une seconde. Enfin, la vérification de la conformité du contrat $\mathcal{C}$ est conforme à l'exigence a également pris une seconde.

Ce modèle système issu de l'industrie montre comment notre approche peut s'appliquer. Il fournit également un bilan positif à l'égard des résultats de la vérification. Nous pouvons affirmer que les modèles précédemment invérifiables peuvent être maîtrisés par la méthode de conception et vérification à base de contrats décrite dans cette thèse.

7 Conclusion et perspectives

Pour faire face à la complexité croissante des systèmes temps-réel critiques, nous avons considéré dans cette thèse une méthode compositionnelle de développement à base de composants dirigée par les exigences. Afin d'obtenir directement un modèle correct du système à partir des exigences, nous avons défini un environnement intermédiaire à base de contrats qui permet de spécifier, de façon abstraite, comment une exigence peut se décomposer par rapport aux composants et comment les composants contribuent à sa satisfaction. En conséquence, le contrat est défini par une paire (hypothèse, garantie) ; il est impliqué dans trois relations de raffinement qui permettent la conception itérative, à savoir : la conformité vérifie qu'un contrat raffine une exigence globale du système, la dominance vérifie qu'un contrat plus général est raffiné par un ensemble de contrats plus spécifiques et l'implémentation vérifie qu'un composant raffine son contrat. Ces notions génériques sont structurées dans une méthodologie de raisonnement à base de contrats dans [143, 144, 142] qui est instanciée tout au long de cette thèse.

Nous avons développé un framework à base de contrats comportementaux pour des modèles système conçus avec SysML, qui peut également être utilisé dans la vérification compositionnelle de la satisfaction des exigences de sûreté temporisées. Notre contribution a deux dimensions : d'abord, nous avons introduit le framework générique à base de contrats en SysML en définissant sa syntaxe et d'autre part, nous avons défini sa sémantique par une variante des automates temporisés entrée/sortie et nous avons introduit une méthode de vérification basée sur le model-checking afin de prouver que les relations de raffinement modélisées sont satisfaites.

Dans l'ensemble, la théorie à base de contrats présentée vise à offrir les propriétés suivantes :

- extensibilité : la méthode peut s'appliquer à des systèmes de grande taille et produire une réponse concernant la satisfaction d'exigences globales,
- prévisibilité : les erreurs de conception peuvent être détectées dès les premières étapes de conception et la méthode peut s'appliquer à des modèles architecturaux grossiers pour obtenir des modèles architecturaux plus fins, et
- réutilisabilité : à la fois en conception, par exemple lorsque les contrats sont définis par des instances de type réutilisables, et en vérification, un couple composant/contrat peut être remplacé par un autre tant que le raffinement est satisfait localement.

En outre, cette méthode permet un développement itératif et incrémental des modèles système.

Contrats en SysML

En premier lieu, étant donné que SysML est un langage de modélisation riche, nous avons sélectionné un sous-ensemble de ses éléments de modélisation suffisant pour décrire des systèmes hiérarchiques à base de composants. De plus, étant un langage de modélisation semi-formel qui laisse ouverts plusieurs points de variation sémantique nous nous sommes appuyés sur le profil OMEGA afin d'avoir une démarche de modélisation système rigoureuse en imposant la satisfaction d'un ensemble de règles de bonne formation concernant la sûreté du typage. Ce langage de composants est étendu par du temps continu et la notion d'horloge, et un mécanisme de formalisation et de vérification des exigences de sûreté avec observateurs.

Ensuite, nous avons introduit les notions liées aux contrats en définissant un méta-modèle enrichi par des règles de bonne formation afin de garantir qu'un modèle étendu par des contrats est sans ambiguïté et fortement typé. Le méta-modèle est décrit en utilisant UML et, par conséquence, l'extension proposée peut s'appliquer à n'importe quel modèle UML/SysML. La définition syntaxique du framework à base de contrats est suffisament générique pour être utilisée avec d'autres formalismes sémantiques, à condition qu'ils expriment la sémantique d'un modèle SysML. Notre définition de contrats gère explicitement un aspect important de la conception dirigée par les exigences : les composants/contrats peuvent avoir une signature plus riche que leur version abstraite, soit en incorporant plusieurs exigences dans une implémentation soit en rendant plus explicite la contribution du composant pour la satisfaction d'une exigence.

Nous avons instancié le méta-modèle pour le langage à composants OMEGA en utilisant le mécanisme des stéréotypes afin de le rendre utilisable dans les éditeurs de modèles et nous avons modélisé les règles de bonne formation définies en utilisant OCL afin que les modèles système étendus par des contrats soient statiquement vérifiables.

Afin de garder la description simple, nous avons supposé certaines restrictions sur le modèle de composants qui n'ont pas d'impact sur l'expressivité d'un modèle : toutes les communications transitent par les ports et un port peut être typé seulement par une interface. Des travaux futurs peuvent explorer l'assouplissement de ces conditions. Par exemple, permettre en SysML le typage des ports avec plusieurs interfaces afin d'offrir plusieurs fonctionnalités pour plusieurs composants au même point d'interaction. Le raffinement de la signature du contrat sera alors transféré de la définition des ports à la définition du type des ports. En effet, il est intéressant de permettre à l'utilisateur de redéfinir le type d'un port modélisé pour la garantie en considérant seulement un sous-ensemble. Pourtant, une telle modélisation transférerait la complexité du processus de conception des ports aux types et pourrait surcharger la compilation d'une relation de dominance, car la cible des signaux devrait être automatiquement calculée en prenant en compte les ports et les connecteurs et pour laquelle des vérifications de typage s'imposent.

Théorie formelle à base de contrats pour les automates temporisés entrée/sortie

La nouveauté de cette notion de contrat est donnée par l'aspect comportamental de l'hypothèse/garantie qui exprime des propriétés sur la dynamique d'un composant via une machine à états. Par conséquent, vérifier les relations de raffinement (comportamental) dans lesquelles les contrats sont impliqués exige de formaliser la sémantique du langage à composants. Le deuxième objectif de cette thèse consiste à définir un cadre sémantique en terme d'automates temporisés entrée/sortie afin d'incarner le langage à base de composants étendu par des contrats. Nous avons défini un framework d'automates temporisés entrée/sortie en nous basant sur [108] respectant la sémantique du modèle de composants et nous avons établi comme relation de raffinement entre les composants l'inclusion des traces temporisées préservée par la composition. Par la suite, nous avons présenté la correspondance entre les éléments de modélisation SysML et les automates temporisés entrée/sortie et nous avons esquissé un algorithme pour la génération des obligations de preuve.

A partir des composants représentés par des automates temporisés entrée/sortie, nous avons construit le framework à base de contrats en définissant la sémantique des obligations de preuve, c.à.d le raffinement dans un contexte sur lequel la satisfaction d'un contrat et la dominance reposent. A son tour, le raffinement dans un contexte repose sur la relation de conformité, qui est définie dans notre cas comme l'inclusion des traces temporisées. Dans ce contexte à base de contrats, le raisonnement pour les exigences de sûreté temporisées est à la fois bien défini et correct, ce qui est établi par les propriétés compositionnelles que la théorie satisfait, c.à.d le raffinement dans un contexte est préservé par la composition et garantit la correction du raisonnement circulaire.

Parce que les obligations de preuve consistent à vérifier un ensemble de relations d'inclusion dee traces temporisées et l'inclusion des traces est indécidable (sauf pour certaines catégories d'automates temporisés), nous avons présenté une méthode de vérification pour une sous-classe de propriétés de sûreté qui permet de décider l'inclusion des traces par model-checking. Ainsi, une garantie est transformée en un automate de propriété temporisé (un observateur) et nous prouvons que cette transformation est suffisante pour vérifier l'inclusion de traces. Cependant, la méthode de vérification est limitée aux propriétés de sûreté déterministes. Une attention particulière doit être accordée aux hypothèses, qui, différemment des garanties, ne sont pas tenues d'être modélisées par des propriétés de sûreté par le framework formel. Rappelons que les hypothèses jouent le rôle d'exigences de sûreté temporisées lorsqu'on prouve la dominance. Par conséquent, modéliser les hypothèses comme des exigences de sûreté temporisées peut s'avérer difficile surtout si le comportement de l'environnement qu'elles abstraient contient des gardes temporisées strictes sur les actions.

Les travaux futurs concernent le renforcement du type d'exigences qui peuvent être vérifiées par l'approche à base de contrats. La motivation est exprimée par le langage qui est utilisé afin de modéliser des exigences de sûreté et qui ne permet pas de spécifier des gardes temporisées strictes. Une idée serait d'utiliser la simulation temporisée comme relation de conformité et/ou raffinement dans un contexte, celle-ci permettant d'exprimer des contraintes temporelles plus fortes afin de se dispenser de la condition de fermeture sous l'extension du temps : le composant abstrait observe au moins le même écoulement de temps que le composant concret et les sorties/actions visibles ne seront plus typés *lazy*. De même, une relation de simulation permettrait d'étendre le type d'exigences vérifiables avec, par exemple, le progrès. Un exemple de relation de raffinement dans un contexte basée sur la simulation qui vérifie des exigences de sûreté et de progrès et qui garantit la correction du raisonnement circulaire est donné en [99, 98] pour le framework BIP. Deux questions sont cependant soulevées par l'utilisation de la simulation : (1) comment le raffinement de signature peut-il être pris en compte et quelle sémantique est à utiliser pour les actions qui ne sont pas explicitement modélisées dans la garantie et (2) comment peut-on automatiquement vérifier la simulation ? En ce qui concerne le premier point, nous avons considéré un raffinement de signature covariant (c.à.d plus d'entrées et de sorties dans le composant), tandis que les actions ne figurant pas dans la garantie sont ignorées. Cependant, d'autres interprétations sont possibles : un raffinement de signature co- et contravariant (c.à.d plus d'entrées, moins de sorties dans le composant) peut être envisagé, alors que les actions qui ne sont pas explicitement modélisées dans la garantie peuvent être assimilées à des erreurs. En ce qui concerne le deuxième point, intégrer plusieurs types de besoins dans une relation de simulation nécessite de la personnaliser, ce qui risque d'induire une complexité croissante de la vérification automatique si elle n'est pas équipée d'un outil personnalisable.

Mise en œuvre et retour d'expériences

L'approche à base de contrats est partiellement mise en oeuvre dans la boîte à outils IFx2. Le compilateur *uml2if* transforme automatiquement un modèle système étendu par des contrats dans un réseau d'automates temporisés entrée/sortie, qui peut être vérifié par model-checking et simulé avec les outils IFx2. Les relations de raffinement modélisées, ainsi que les contrats, ne sont pas encore pris en charge pour la génération des obligations de preuve, ces étapes restant manuelles. Pour nos expérimentations, nous avons manuellement modélisé différentes variantes du système qui correspondent aux membre gauches des relations de conformité obtenues, ainsi que la formalisation d'une hypothèse/garantie modélisée comme un composant dans une exigence de sûreté. Les travaux futurs concernent l'automatisation de toutes les étapes intermédiaires décrites et l'ajout de la gestion des obligations de preuve.

Enfin, nous avons illustré notre méthode sur deux études de cas — un exemple paramétrique et un autre extrait d'un modèle système d'échelle industrielle — et nous avons montré comment notre approche peut atténuer le problème d'explosion combinatoire de l'espace d'états pour la vérification de grands systèmes. Cette affirmation est confirmée par les résultats positifs obtenus pendant la vérification.

Nous concluons que notre framework à base de contrats est approprié à la conception et la vérification compositionnelle des systèmes critiques temps-réel de grande taille décrits par plusieurs couches architecturales, qui doivent satisfaire des exigences de sûreté temporisées et qui ne modélisent pas de contraintes temporelles fortes pour les actions.

Génération automatique des contrats

Une question importante qui n'a pas été traitée dans cette thèse et qui constitue des travaux futurs concerne le guide méthodologique pour modéliser des contrats. L'absence d'une méthode bien définie qui décrit comment obtenir/modéliser les contrats pour le système entier et les composants est une des raisons de la difficulté à exploiter les contrats en génie logiciel. En particulier les langages de programmation n'ont pas eu le succès escompté. Néanmoins, nous pensons que le cas des contrats dans les premières phases de l'ingénierie système est différent et que la notion de contrat est fortement nécessaire pour décomposer les systèmes et pour vérifier leur correction avant l'implémentation. Nous plaidons pour l'application de techniques de vérification, même si leur coût est assez important, peut être dans la plupart des cas moindre qu'identifier des erreurs dans le système après implémentation et déploiement. Par conséquent, fournir des méthodes ou des guides méthodologiques pour dériver des contrats intermédiaires à partir des exigences qu'on veut prouver est notre perspective à long terme.

Une idée serait de générer automatiquement les garanties à utiliser en se basant sur les exigences locales. En effet, à partir d'une exigence globale nous pourrions déterminer un premier contrat qui intègre dans sa signature des actions spécifiques (visibles) qui peuvent être utilisées par la suite dans la décomposition du contrat. En général, ce contrat devrait être spécifié par le concepteur. Ensuite, à partir de la garantie nous pourrions projeter la garantie globale sur un ensemble de sous-garanties en tenant compte de l'hypothèse globale et de la signature de chaque sous-composant. La correction des garanties locales par rapport à la garantie abstraite peut être assurée avec une approche CEGAR. Nous supposons que cette étape peut être appliquée de façon itérative jusqu'à ce que le niveau de granularité atteint corresponde à une implémentation du contrat. Une telle approche a récemment été étudiée [96] pour des systèmes à transitions étiquetées sous le concept de *réalisabilité*.

Ce raisonnement correspond à la technique que nous avons appliquée pour déterminer les contrats tout au long des études de cas de cette thèse. Pour chaque exemple, nous avons commencé par identifier les actions que le composant exécute et qui contribuent directement à la satisfaction du besoin, mais aussi celles dépendantes de l'environnement. Les garanties modélisées sont les plus faibles composants qui simulent simplement la structure de leurs implémentations correspondantes. Pour l'étude de cas industrielle, la tâche a été simple et directe : la garantie globale est presque identique à l'exigence, tandis que les garanties des composants représentent la projection de la garantie globale sur les composants. Pourtant, cette étape nécessite de spéculer sur une architecture à base de composants réalisable et un ensemble d'actions possibles.

Dans la même ligne de pensée, nous nous intéressons à la génération automatique des hypothèses sur l'environnement qui doivent être modélisées pour chaque contrat. Le raisonnement est similaire à la génération de la plus faible pré-condition pour un triplet de Hoare : la garantie G_i a été précédemment calculée, soit automatiquement, soit par définition, et l'implémentation K_i est donnée ; alors un environnement A_i tel que $K_i \sqsubseteq_{A_i} G_i$ peut être déterminé. Si A_i ne peut pas être calculé, cela signifie que K_i n'est pas une implémentation correcte pour G_i. Nous pouvons aussi utiliser les obligations de preuve que la relation de dominance génère, si l'implémentation K_i n'est pas fournie : G, G_i et A sont connus ; nous devons alors déterminer le plus faible (et simple) A_i tel que la relation de conformité écrite $A \parallel (\parallel_{j=1}^{n} G_j) \preceq A_i \parallel G_i$ soit satisfaite. Comment calculer A_i en se basant sur la satisfaction de contrat a été étudié [84, 134, 83] pour les systèmes à transitions étiquetées avec CEGAR et l'apprentissage automatique.

Ces résultats fournissent une technique entièrement automatisée pour concevoir des modèles système corrects par construction, qui pourrait être intégrée dans un processus itératif de développement à base de composants.

Diagnostic d'erreurs avec les contrats

A court terme, une seconde perspective consiste à explorer le diagnostic d'erreurs pour le framework à base de contrats. En effet, transférer le contre-exemple obtenu sur le modèle formel dans le modèle semi-formel permettra aux ingénieurs d'identifier plus facilement l'erreur et la corriger. Avoir une transformation bidirectionnelle traçable entre les deux modèles représenterait un atout pour la modélisation des abstractions — sous la forme d'hypothèses/garanties dans notre cas — et leur correction, chaque fois que le raffinement n'est pas satisfait localement.

De plus, le diagnostic doit être effectué sur l'arbre de preuve en entier, et pas seulement localement. Nous avons esquissé à la section 5.4 un ensemble d'indications de ce qui est envisageable si l'on ne parvient pas à prouver la satisfaction, ce que nous avons appliqué aux études de cas présentées pour définir les abstractions correctes. Cependant, ces règles doivent être soigneusement étudiées et, finalement, définies comme méthodologie de diagnostic, ce qui pourrait être intégré au processus de développement actuel.

Introduction

Context and motivation

Error-free safety critical systems are hard to develop and several recent disasters[2] support this statement. For example, the Ariane 5 flight 501 exploded at 37 seconds after launch[3] due to an incorrect data conversion causing the loss of more than 370 million dollars, the MIM-104 Patriot anti-ballistic missile[4] failed to intercept a missile, which killed 28 soldiers, due an internal clock drift that resulted in wrong computations for projectile searching and tracking and the Multidata Systems' Cobalt-60 radiation therapy machine[5] overdosed several dozens of patients in 2000 due to incorrect calculations dependent on the data entry sequence. System errors (or bugs) are responsible for important casualties going through security issues, environmental disasters and even loss of human life. Ensuring the safety and correctness of such critical systems' behavior is essential. A particular case is represented by real-time systems for which, besides the order of actions or their carried values, the correctness relies also on their timing, i.e. when an action must be performed.

The development of critical real-time embedded systems is a challenging task. There are two key factors that need to be considered during development: (1) what is the best method for designing large and complex systems with minimal effort and costs and (2) how can it be ensured that the designed system is correct with respect to its requirements. Indeed, as systems grow in size and complexity so is the number of errors they contain and which become harder to identify and correct, while their impact on the final product may have catastrophic consequences.

In order to master the system's size, system engineers have adopted a compositional

[2] http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html

[3] http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm

[4] http://archive.today/XbB5

[5] http://www.iaea.org/newscenter/features/radiotherapy/dissection109.pdf

design method based on components which allows to recursively decompose the problem to solve, typically a requirement, into smaller ones (a divide-and-conquer approach), until the desired granularity level is achieved. In consequence, designers will deal with small specifications which are easier to develop, called atomic components, or with the assembling of components by composition which results in a hierarchical component. This design method has several advantages: separation of concerns from the system's decomposition, incremental development by successive refinements, independent implementation of components by different engineering teams and reusability that is harder to achieve.

Yet, having multiple suppliers building integrated systems based on common requirements entails a risk of errors during the development due to the difficulty of decomposing global system requirements on components and the misinterpretation of system requirements allocated to the software. Therefore, the limitation of this method resides in the inherent compositional aspect: it is difficult to design a network of components that satisfy by their interaction a global requirement, while, furthermore, components are usually involved in the satisfaction of several requirements.

With respect to the second expressed issue, we need to take into consideration the fact that early designs are often realized using semi-formal languages like UML [91], SysML [90] or AADL [148] which lack a proper mechanism for formalizing requirements and proving their satisfaction. The errors potentially introduced during development are then discovered late and by very costly processes. Therefore, the last decades have seen an accelerating utilization of formal verification and validation techniques in the early phases of the development process in order to guarantee as soon as possible the correctness of the design, as well as reducing production costs and increasing system quality. From a correct design model via proven refinement relations we can obtain a correct system implementation which can be immediately deployed. It implies that formal methods can be used to derive correct-by-construction implementations from high-level specifications if integrated in an iterative development process.

Design models are validated using an assortment of techniques, including design review [132], testing, interactive simulation and model-checking [141, 49, 52]. The first three methods allow to detect errors in a lightweight way since they explore only a subset of the system's behaviors and, in consequence, they do not guarantee the correctness of the system with respect to requirements. Model-checking is a fully automated technique that exhaustively explores system's behaviors, which are represented by a state space model. The state space is usually a finite graph.

However, for very large systems the state graph cannot be completely computed, which makes model-checking suffer from the state space explosion problem: the system becomes soon intractable if components execute concurrently, because the number of states grows exponentially in size with the number of components. Therefore, a verdict about requirement satisfaction cannot be provided. Examples [71, 40, 21] show that common verification techniques may find themselves powerless in front of the complexity of industrial-grade systems.

Three types of improvements have been studied in the literature:

1. reducing the state space by modifying the mathematical representation,
2. modeling abstractions for system's components and verifying the requirement satisfaction on the abstract model, and
3. decomposing the global requirement to properties that need to locally hold on components, i.e. compositional approach.

We consider that the first approach is only partial if it cannot sufficiently reduce the complexity of system designs in order to make them tractable. The last two approaches are dual, since the specification plays the role of the component's abstract behavior for the second, and a constraint over the component's behavior for the third. However, their main limitation is that they do not take into account the behavior of the component's environment and, therefore, the component needs to correctly refine its abstraction/constraint independently. This can be hard to achieve since between a component and its environment there are usually mutual dependencies on which their correctness is based. In consequence, the environment, which can suffer from the state space explosion problem on its own, needs to be considered during refinement.

In this thesis we propose to combine the abstract and compositional techniques and define a unique specification for a component that is both abstract and partial. The environment should also be constrained by such a specification. Therefore, we use the notion of *contract* for a component, defined as a pair (assumption, guarantee) where the *assumption* is an abstraction of the environment's behavior and the *guarantee* is the component's abstract behavior with respect to the running requirement, given that the environment behaves like the assumption. The assumption is correct if the environment refines it in the abstract context of the guarantee. This type of reasoning with contracts is called circular.

Informally, a contract models the point of view of the component and its contribution toward the satisfaction of one requirement. Then, one component may implement several contracts, one for each requirement that has to be satisfied. The set of contracts corresponding to the network of components needs to correctly assemble and satisfy the requirement. Hence, the number of relations to be verified in order for contract-based reasoning to hold is linear in the number of designed components and we can assume that the involved compositions are in general reduced and can be handled by automatic verification tools.

Contracts are a valuable asset for correct-by-construction component-based design since they can be used to:

1. constrain the component's behavior with respect to one requirement, while several contracts can be integrated in the same implementation,
2. substitute and reuse (off-the-shelf) components that satisfy the given contract,
3. independently implement components based on the given contract without challenging the requirement satisfaction, and
4. iteratively design systems by proven refinement.

These features are supported by the three refinement relations a contract can be subject to: *conformance* verifies that the contract satisfies a requirement, *dominance* verifies refinement between contracts and *implementation* verifies that a component satisfies its contract. Moreover, contract-based reasoning offers diverse opportunities: mapping and tracing requirements to components, tracking the evolution of requirements during development, aid in model reviews, virtual integration of components and, most importantly, compositional verification.

The contract-related notions presented above have been defined in [143, 144, 142] in the form of a *meta-theory*. By meta-theory we denote a generic contract-based framework that describes how the reasoning can be applied in system design and compositional verification, but without providing a precise definition for these concepts. In order to obtain a working framework for a specific component model, one has to formalize the component framework — define at least the notions of component, composition and refinement — and the contract framework — define conformance, dominance and satisfaction. Moreover, a set of compositionality properties must be proved to hold in the instantiation in order to guarantee the soundness of the reasoning.

Contribution

Despite the obvious benefits, systems engineering does not use contract-based reasoning as development technique for system models described with semi-formal languages (e.g. UML, SysML, etc.) due to the lack of definition of a sound and complete contract-based framework directly applicable to such designs. The aim of this thesis is to graft contract-based reasoning in the model-driven design and requirement verification process for critical real-time embedded systems described with SysML. To the best of our knowledge, this study is the first to link high-level modeling languages and formal behavioral contracts.

Our contribution is manifold. In order to use contracts as first-class elements in SysML, we define the syntax of the contract-related notions via a UML-based meta-model. A set of well-formedness rules accompanies the meta-model in order to ensure its compliance with the sound contract-based methodology described in [143, 144, 142]. For example, such rules cover the actions a contract/component may perform and their refinement by integrating several constraints into a component. We instantiate the meta-model in the context of the OMEGA profile such that it can be used by model editors. OMEGA [87] is a UML/SysML specialization, which allows for rigorous engineering of real-time systems by providing a connection with the IFx verification and validation toolset [34].

Secondly, we formalize the semantics of the SysML component language extended with contracts by a variant of Timed Input/Output Automata (TIOA). This transformation opens up the possibility to formally verify that the refinement relations modeled between contracts and/or components hold, which implies the satisfaction of the global requirement. In consequence, we define the mapping of concepts from SysML to TIOA and we sketch how to explore a model in order to generate the proof obligations corresponding to the desired refinement relations. We partially implement this transformation in the IFx2 toolset with a compiler, which takes as input a system design modeled with OMEGA in the XMI 2.0 format and produces the network of TIOA on which the verification will be performed. The updated version of the compiler extends the IFx2 features, namely simulation and model-checking of real-time systems, to system models compliant with UML 2.3/SysML 1.1.

Based on the formal component framework we build the contract-based theory by defining a proof obligation for each type of refinement relation. The obtained contract-based framework is an instantiation of the meta-theory defined in [143, 144, 142], where components are represented as TIOA and proof obligations are defined by a timed trace inclusion relation that takes into account the environment and is denoted hereafter refinement under context. We prove that refinement under context is preserved by composition and ensures sound circular reasoning, two important results which are a prerequisite of the meta-theory guaranteeing the soundness of the method. Since timed trace inclusion is undecidable and therefore cannot be automatically verified, we propose to use model-checking on each proof obligation where the local requirement is transformed in a formal *timed safety property*. Informally, a safety property models that something bad (i.e. undesirable) never happens during the system's execution. We prove that this transformation and the application of reachability analysis is sufficient to guarantee, in our case, timed trace inclusion. Since the formal framework and the verification method impose some restrictions with respect to the expressiveness of contracts, we discuss their impact on the component language used for modeling contracts and we define a particular timed semantics which satisfies by default the required restrictions, i.e. the guarantee must be a (deterministic) timed safety property.

Finally, the approach has been evaluated on two case studies among which one consists in an industrial-grade system design, the Solar Generation Wing System (SGS) of the Automated Transfer Vehicle (ATV), for which we verify of a general safety requirement. The obtained results are encouraging: for the SGS case study, the contract-based approach required the effort of 5 person*days for contract modeling and performing verification, while monolithic model-checking, even in conjunction with reduction techniques, fails to provide a result.

The results obtained in this thesis have been published in the following list of papers:

1. Iulia Dragomir, Iulian Ober, and Christian Percebois. *Safety Contracts for Timed Reactive Components in SysML.* In 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), 25/01/2014–30/01/2014, Novy Smokovec, Slovakia, pages 211-–222. Springer, January 2014.
2. Iulia Dragomir, Iulian Ober, and Christian Percebois. *Integrating Verifiable Assume/Guarantee Contracts in UML/SysML.* In 6th International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES- MB), Miami, USA, 29/09/13–29/09/13. CEUR Workshop Proceedings, November 2013.
3. Iulia Dragomir, Iulian Ober, and Christian Percebois. *Safety contracts for reactive timed systems (extended abstract).* In Action AFSEC, Journées GDR GPL, 02/04/2013–05/04/2013, Nancy, France, pages 37–46, April 2013.
4. Iulia Dragomir, Iulian Ober, and David Lesens. *A Case Study in Formal System Engineering with SysML.* In 17th International Conference on Engineering of Complex Computer Systems (ICECCS), 18/07/2012–20/07/2012, Paris, France, pages 189–198. IEEE, July 2012.
5. Eric Conquet, François-Xavier Dormoy, Iulia Dragomir, Susanne Graf, David Lesens, Piotr Nienaltowski, and Iulian Ober. *Formal Model Driven Engineering for Space Onboard Software.* In International Conference on Embedded Real Time

Software and Systems (ERTS2), 01/02/2012–03/02/2012, Toulouse, France. SAE, January 2012.

6. Ileana Ober, Iulian Ober, Iulia Dragomir, and El Arbi Aboussoror. *UML/SysML semantic tunings.* Innovations in Systems and Software Engineering, 7(4):257–264, 2011.
7. Eric Conquet, François-Xavier Dormoy, Iulia Dragomir, Alain Le Guennec, David Lesens, Piotr Nienaltowski, and Iulian Ober. *Modèles système, modèles logiciel et modèles de code dans les applications spatiales.* Génie logiciel, 1(97):9–15, juin 2011.
8. Iulian Ober and Iulia Dragomir. *Unambiguous UML composite structures: the OMEGA2 experience.* In 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), 22/01/2011–28/01/2011, Novy Smokovec, Slovaquie, pages 418–430. Springer, January 2011.
9. Iulia Dragomir and Iulian Ober. *Well-formedness and typing rules for UML Composite Structures.* CoRR, abs/1010.6155, October 2010.
10. Iulian Ober and Iulia Dragomir. *OMEGA2: A new version of the profile and the tools.* In UML&AADL'2010–15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 24/03/2010–25/03/2010, Oxford, UK, pages 373–378. IEEE, March 2010.

The contribution is also supported by two internal research reports:

11. Iulia Dragomir, Iulian Ober, and Christian Percebois. Integrating verifiable Assume/Guarantee contracts in UML/SysML. Technical Report IRIT/RT- 2013-14-FR, IRIT, July 2013.
12. Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety Contracts for Timed Reactive Systems. Technical Report IRIT/RT-2013-11-FR, IRIT, June 2013.

Organization

This thesis is structured in three parts, as follows: Part I focuses on the motivation and the context of our work by describing the limits of current design and verification techniques, Part II presents the theoretical contribution of our work, the instantiation and implementation of the selected contract-based meta-theory for systems described with SysML, and Part III describes the practical contribution and assesses the verification results of our method with respect to monolithic model-checking.

The first part contains three chapters that tackle the state of the art for both formal and semi-formal system design and verification, respectively contract-based reasoning:

- Chapter 1 presents a survey of the formal modeling languages, that can be used to represent critical real-time systems, and of the verification techniques, which can be used to guarantee their correctness. We present the basic notions about Timed Input/Output Automata and Timed Transition Systems, which are used throughout this thesis to describe the semantics of the developed systems. With respect to the verification aspect, we focus on model-checking as it is technique that we adopt for our contract-based framework.
- Chapter 2 presents a survey of high-level modeling languages and their associated profiles used in academia and industry for designing real-time embedded systems. We inventory a non-exhaustive list of the validation and verification environments built for such semi-formal designs, where model-checking tools play an important role.
- Chapter 3 describes the method of reasoning with contracts in the form of the meta-theory which is instantiated in this thesis. The chapter covers the related work with respect to other defined meta-theories and their instances, as well as the notion of contract as it is used in high-level modeling languages.

The second part details our theoretical contribution. More specifically:

- Chapter 4 sets out the context of our contributions by presenting the set of notions from UML/SysML which we use throughout this thesis for modeling real-time systems. We introduce the OMEGA profile whose aim is to provide a clear and coherent syntax and semantics for UML/SysML and which constitutes our working context.
- Chapter 5 defines the syntax for the contract-related notions described by the meta-theory in the form of a UML meta-model and the set of well-formedness rules which ensures the compliance of any model extended with contracts to the meta-theory. We instantiate the domain meta-model for the OMEGA component language and we show how the contract methodology can be applied on a precise system design.
- Chapter 6 defines our contract-based theory by formalizing the semantics of a system design extended with contracts with Timed Input/Output Automata and specifying the proof obligation generated by each refinement relation modeled between contracts and/or components. We show that the reasoning with contracts is sound by proving that the required compositionality results hold in our framework. We present a model-checking based verification algorithm which transforms a timed input/output automaton into a timed property automaton that formalizes a safety property. We discuss the limitations that

the formal framework imposes on the expressiveness of contracts in SysML.
- Chapter 7 presents the implementation of our framework in the IFx2 Toolset. We sketch how the modeling concepts can be mapped between (OMEGA) SysML to TIOA and we present a proof generation algorithm for the contract-related refinement relations modeled in a design. We describe the functionalities of the toolset, e.g. how to use contracts to debug erroneous models.

The third part evaluates the advantages of the contract-based approach on two case studies:

- Chapter 8 presents a parametric case study whose aim is to show that our approach is beneficial even for small systems that generate a large state space.
- Chapter 9 describes a case study extracted from the ATV real-life system, which shows how the contract-based framework can be used to alleviate the state space explosion problem and provide a yes/no requirement satisfaction result where monolithic model-checking fails.

State of the Art Part I

1 Formal Modeling and Verification of Real-Time Embedded Systems: Current Approaches

System modeling plays a key role in the development process since it allows to represent an abstraction of the system by focusing only on the crucial aspects, both functional and non-functional. Integrated in a model-driven development approach, such a model can be repeatedly refined towards implementations, while incorporating formal frameworks will guarantee the correctness of the obtained system either by construction or by applying verification and validation techniques. Therefore, errors potentially introduced during design are discovered in the early phases of the development with less costs and reduced consequences.

This thesis is at the crosswalk between formal methods and model-driven systems engineering in the context of critical real-time embedded systems (RTES). In this chapter we focus on the formal aspect by presenting a survey of the modeling frameworks, as well as of the verification techniques currently used at system development.

Section 1.1 presents the state of the art in the formal modeling of timed reactive systems with a particular interest for Timed Input/Output Automata and Timed Transition Systems for which we recall some basic definitions. Timed Input/Output Automata represent the formal base of our work, while the latter describe the semantics of our models. In Section 1.2, we provide a discussion of the verification methods that can be applied on (formal) models. The comparison of their advantages and drawbacks shows the motivation to implement a verification technique with contracts for system designs.

1.1 Formal Models for Reactive Systems

Our interest focuses on the correct development of RTES where the central role is played by the system design and the requirements it must satisfy. Typically, a development process (e.g. the V-process) starts with a coarse-grained architecture derived directly from the system requirements, which is then subject to iterative decomposition and stepwise refinement until the needed level of granularity is reached. Then at each refinement step, the obtained design has to be correct with respect to the specified requirements. In order to evaluate whether such constraint is satisfied we make use of *formal models*, which allow us to precisely represent the system model (including its behavior) and the requirements it must satisfy. Verification techniques, described in Section 1.2, developed on top of formal models can provide an answer with respect to the satisfaction of system requirements.

However choosing the appropriate formal model to represent system designs is not an easy task. It depends on several parameters like the elements that need to be modeled and the type of requirements the system has to satisfy. Some formalisms are too rich in what they model, e.g. modeling languages used in industry do not usually capture the probabilistic aspect of actions which is considered by the probabilistic system framework. While other formalisms need to be extended in order to cover all aspects of a system, e.g. the input/output directionality of RTES actions is defined as an extension of classical formal models. In consequence, each modeling framework is targeted for the verification of certain types of requirements. We discuss in the following the formal models defined in the literature that can be used to represent the syntax and semantics of high-level modeling languages, while we limit our attention to general safety requirements, i.e. something bad does not happen. These languages like UML [91] and lately SysML [90] are often used in system engineering for system design since they provide an intuitive set of concepts as modeling elements. Such designs describe mainly a reactive model of computation, i.e. event-driven: an action executed by the system is the effect of a previous event. Therefore, in the following, we will focus on models that exhibit or can incorporate an asynchronous communicating semantics. We mention that SysML introduces elements that have a time-triggered model of computation such as synchronous architectures, yet this communication paradigm has been explored elsewhere and is not of concern.

For each framework we consider the main features needed to represent system designs from high-level modeling languages: non-deterministic and timed behavior, composition operator for hierarchical design of systems and refinement mechanisms between systems at different abstraction levels.

1.1.1 Modeling Semantics: Transition Systems

Transition systems are abstract machine models used to describe the behavior of systems by means of states and transitions, while they are illustrated as a directed graph. They are the basic notation to represent the semantics of any type of model.

The *state* of a system models its precise features at a particular instant during its execution. The state changes by firing *transitions* which model how the system evolves. Each transition has a source state and a target state. Transitions can be labeled with the name of the action performed by the system. Such systems are called *labeled transition systems.*

Definition 1.1 (Labeled transition system). A *labeled transition system* (LTS) $\mathcal{S}$ is a 4-tuple $(Q,\ \theta,\ A,\ D)$ where:

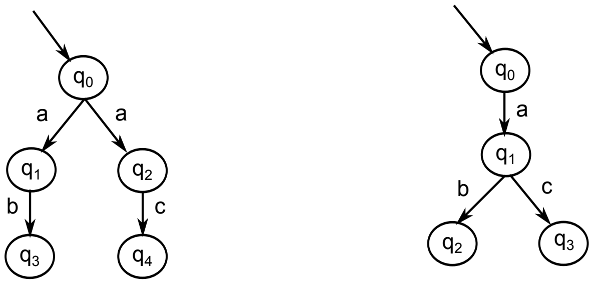
- Q is a non-empty set of states.
- θ is an initial state.
- A is the set of labels. This set contains the name of the observable actions and ε, an internal silent action.
- $D \subseteq Q \times A \times Q$ is the transition relation.

Notation. We often denote the elements of an LTS $\mathcal{S}$ by $Q_\mathcal{S}$, $\theta_\mathcal{S}$, etc. We omit these subscripts where no confusions seems likely. For a set of systems $\mathcal{S}_i$ we use as subscript the index i. Any transition $(q, a, q') \in D_\mathcal{S}$ is denoted by $q \xrightarrow{a}_\mathcal{S} q'$. Again we drop the subscript when $\mathcal{S}$ is clear from the context, or we use the index i for a set of systems. We call q the source or origin state of the transition and q' the target or destination state. If for a state q and an action a there is a state q' such that $q \xrightarrow{a} q'$ then we say that a is *enabled* in q.

Figure 1.1(a) represents the LTS $\mathcal{S}_1 = (Q_1,\ \theta_1,\ A_1,\ D_1)$, where $Q_1 = \{q_0, q_1, q_2, q_3, q_4\}$, $\theta_1 = q_0$, $A_1 = \{a, b, c\}$ and $D_1 = \{q_0 \xrightarrow{a}_1 q_1, q_0 \xrightarrow{a}_1 q_2, q_1 \xrightarrow{b}_1 q_3, q_2 \xrightarrow{c}_1 q_4\}$. States are denoted by circles that contain their name, while transitions are marked by arrows. The action performed by a transition is written on the arrow. The initial state is represented using an arrow with no origin state.

Definition 1.2 (Determinism). An LTS $\mathcal{S}$ is *deterministic* if for every state q and every label $a \in A \setminus \{\varepsilon\}$ there is at most one state q' such that $q \xrightarrow{\varepsilon *}_\mathcal{S} \xrightarrow{a}_\mathcal{S} q'$.

An LTS that is not deterministic is called *non-deterministic.* The LTS $\mathcal{S}_1$ from Figure 1.1(a) is non-deterministic since in the initial state q_0 and for the label a there are two outgoing transitions reaching different states, $\{q_1, q_2\}$. The LTS $\mathcal{S}_2$



(a) Non-deterministic LTS $\mathcal{S}_1$. (b) Deterministic LTS $\mathcal{S}_2$.

Figure 1.1 – LTS examples.

represented in Figure 1.1(b) is, on the other hand, deterministic. Furthermore, $\mathcal{S}_2$ can be obtained via the determinization process from $\mathcal{S}_1$.

Large systems can be obtained from individually developed LTS by applying the parallel composition operator. Informally, two LTS when executed in parallel synchronize on common labels, while distinct actions are interleaved.

Definition 1.3 (Parallel composition). Let $\mathcal{S}_1$= (Q_1, θ_1, A_1, D_1) and $\mathcal{S}_2$ = (Q_2, θ_2, A_2, D_2) be two LTS. $\mathcal{S}_1 \parallel \mathcal{S}_2$ is the LTS (Q, θ, A, D) where:

- $Q \subseteq Q_1 \times Q_2$,
- $\theta = (\theta_1, \theta_2)$,
- $A = A_1 \cup A_2$ and
- $D \subseteq Q \times A \times Q$ is the transition relation generated by the following rules:
    1. if $q_1 \xrightarrow{a} q_1'$ and $a \in (A_1 \setminus A_2) \cup \{\varepsilon\}$ then $(q_1, q_2) \xrightarrow{a} (q_1', q_2)$,
    2. if $q_2 \xrightarrow{a} q_2'$ and $a \in (A_2 \setminus A_1) \cup \{\varepsilon\}$ then $(q_1, q_2) \xrightarrow{a} (q_1, q_2')$ and
    3. if $q_1 \xrightarrow{a} q_1'$ and $q_2 \xrightarrow{a} q_2'$ and $a \in (A_1 \cap A_2) \setminus \{\varepsilon\}$ then $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$.

For the example from Figure 1.1, $\mathcal{S}_1 \parallel \mathcal{S}_2$ builds an LTS having an identical structure as $\mathcal{S}_1$.

An execution of an LTS, which is denoted *path* or *run*, records the actions that the LTS has performed as well as state modifications. Informally, a path starts from the initial state and contains a finite or infinite sequence of states that can be reached via successive actions. Indeed, a reactive system can have an infinite execution called ω-sequence.

Definition 1.4 (Path). A *path* (or *run*) π for an LTS $\mathcal{S}$ is a finite or infinite sequence $q_0 a_1 q_1 a_2 q_2 \ldots$ where $\forall i$, $a_i \in A$, $q_i \in Q$, $q_0 = \theta$ and $q_i \xrightarrow{a_{i+1}} q_{i+1}$.

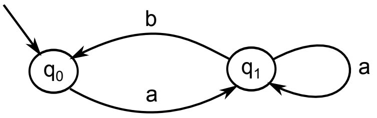


Figure 1.2 – An LTS $\mathcal{S}_3$ having infinite paths.

We can define *path fragments* from a given state q by requiring the sequence to start from q, i.e. $q_0 = q$. We denote the set of path fragments from a state q with *frags*(q). For the LTS $\mathcal{S}_1$, a finite path corresponds to $\pi = q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_3$. An LTS having an infinite path is represented in Figure 1.2. Such a path consists in infinite sequences of a and b, where each b is preceded by at least one a.

A path records all executed actions and state changes. However, we may be interested in keeping from a run only the observable information, e.g. abstracting the execution of the silent action ε. We introduce the notion of trace as the projection of a path on observable actions.

Definition 1.5 (Trace). Let $\pi = q_0a_1q_1a_2q_2a_3\ldots$ be a path of an LTS $\mathcal{S}$. *trace*(π), called *trace* of $\mathcal{S}$, consists of a finite or infinite sequence of labels a_j of π, where all labels $a_i = \varepsilon$ are removed.

We can obtain a *trace fragment* from a state q by requiring that the corresponding path starts from q. We denote the set of traces of an LTS $\mathcal{S}$ by *traces*$_{\mathcal{S}}$ and the set of traces starting in a state q by *tracefrags*$_{\mathcal{S}}(q)$.

The states that are obtained during a run of the LTS are called reachable.

Definition 1.6 (Reachable state). A state $q \in Q$ is *reachable* in $\mathcal{S}$ if there exists a path $\pi = q_0a_1q_1a_2q_2\ldots$ such that $q_i = q$ for some $i \geq 0$.

We denote by *reach*$(\mathcal{S}) \subseteq Q$ the set of states that can be reached via any run of the LTS and by *reach*$(\mathcal{S})(\sigma)$ the set of states reached after the executions corresponding to the trace σ.

Based on the set of traces we introduce trace inclusion, a preorder relation which can be used to define refinement of LTS.

Definition 1.7 (Trace inclusion). Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two LTS. $\mathcal{S}_1$ refines $\mathcal{S}_2$, denoted $\mathcal{S}_1 \preceq \mathcal{S}_2$, if *traces*$_{\mathcal{S}_1} \subseteq$ *traces*$_{\mathcal{S}_2}$.

For the example from Figure 1.1 we have *traces*$_{\mathcal{S}_1} = \{\phi, a, ab, ac\}$ and *traces*$_{\mathcal{S}_2} = \{\phi, a, ab, ac\}$, where ϕ denotes the empty trace (i.e. observable actions have not been executed). Thus, $\mathcal{S}_1 \preceq \mathcal{S}_2$ and $\mathcal{S}_2 \preceq \mathcal{S}_1$.

A second relation which allows to compare LTS by taking into consideration their structure is *simulation* [123].

Definition 1.8 (Simulation). Let $\mathcal{S}_c$ and $\mathcal{S}_a$ be two LTS with $A_c = A_a = A$, where $\mathcal{S}_c$ denotes a concrete system and $\mathcal{S}_a$ an abstract one. A relation $\mathcal{R} \subseteq Q_c \times Q_a$ is a *simulation* relation if and only if $\theta_c \mathcal{R} \theta_a$ and $\forall q_c \mathcal{R} q_a$ and $\forall a \in A$ such that $q_c \xrightarrow{a} q_c'$ then $\exists q_a'$ such that $q_a \xrightarrow{a} q_a'$ and $q_c' \mathcal{R} q_a'$. We write $\mathcal{S}_c \leqslant \mathcal{S}_a$ if there exists such a relation $\mathcal{R}$ between the two LTS.

Informally, $\mathcal{S}_c \leqslant \mathcal{S}_a$ if any reachable state $q_c \in Q_c$ can be mapped to a state $q_a \in Q_a$ (reachable via the same set of labels) and all labels enabled in q_c are also enabled in q_a. We remark that simulation is a reflexive relation, i.e. $\mathcal{S} \leqslant \mathcal{S}$.

For the example in Figure 1.1 we have that $\mathcal{S}_1 \leqslant \mathcal{S}_2$, but $\mathcal{S}_2 \not\leqslant \mathcal{S}_1$ since $q_1^{\mathcal{S}_2} \not\mathcal{R}\, q_1^{\mathcal{S}_1}$: the label c is not enabled in $q_1^{\mathcal{S}_1}$.

The simulation notion defined above corresponds to strong simulation. This relation can be weakened by ignoring the silent action from the mapping and, so, replacing one transition with a sequence as defined by $q \xrightarrow{\varepsilon *}\xrightarrow{a} q'$ for both $\mathcal{S}_1$ and $\mathcal{S}_2$, where $a \in A \setminus \{\varepsilon\}$.

Simulation is strictly stronger than inclusion of traces, as it is described by the following result.

Proposition 1.1. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two LTS. If $\mathcal{S}_1 \leqslant \mathcal{S}_2$ then $\mathcal{S}_1 \preceq \mathcal{S}_2$.*

Proposition 1.1 presents a sufficient condition for deciding trace inclusion by verifying if simulation holds between the concrete and the abstract systems. As we will see, trace inclusion is undecidable for some categories of formal models. Therefore, verifying instead simulation is a good solution. Yet this condition is not necessary: in the example from Figure 1.1 we have that $\mathcal{S}_2 \preceq \mathcal{S}_1$, but $\mathcal{S}_2 \not\leqslant \mathcal{S}_1$.

Several extensions have been developed for transition systems in order to encompass different features: *alternating transition systems* [9] for describing a k-player game, *modal transition systems* [116, 118] for designing both an over and under-approximation of a system by differentiating between *must* and *may* transitions, and *timed transition systems* [101] in order to quantify time elapse, as well as hybrid versions like [8, 38, 27, 26].

An alternating transition system (ATS) is a generalization of labeled transition systems for which a concurrent game structure is considered: a number of distinct agents is defined and, in each state and for each agent, a set of possible transitions is described. At execution, agents are selected either by a scheduler or in a predefined order to perform actions, thus playing a game between them. A particular case is given by the 2-player instantiation corresponding to the system under study and the environment. Then, the system and the environment evolve in parallel and are independent in their choices of transitions (while trying to avoid eventual erroneous behaviors). We consider that ATS are aimed to answer the following question: can the system resolve its internal choices such that the required properties are satisfied regardless of how the environment behaves? Therefore, alternating logic is well suited to model open systems and make the distinction between components and the environment. In a compositional reasoning approach that does not take the environment's behavior into consideration, ATS represent a good semantics for components.

A modal transition system (MTS) makes the difference between transitions that *must* be considered (required transitions) and transitions that *may* be considered (allowed transitions) in a system's implementation. Since these notions do not have an explicit counterpart when modeling RTES with high-level languages such as UML and SysML, we will not consider them in the following as a good candidate to express the semantics of our target RTES.

An important category of transition systems that allow modeling timed behavior are *timed transition systems* (TTS). Basically, a TTS is a labeled transition system where the set of transitions is enriched with transitions labeled with time delay.

Definition 1.9 (Timed transition system). A *timed transition system* (TTS) $\mathcal{S}$ is a 4-tuple $(Q,\ \theta,\ A,\ D)$ where:

- Q is a non-empty set of states.
- θ is an initial state.
- A is a set of labels including the silent action ε.
- $D \subseteq Q \times (A \cup \mathbb{R}_+) \times Q$ is the transition relation.

A TTS satisfies the following axioms, where $\delta, \delta_1, \delta_2 \in \mathbb{R}_+$:

A0) *(0-delay)*
$q \xrightarrow{0} q'$ if and only if $q = q'$.

A1) *(Time additivity)*
$q \xrightarrow{\delta_1} q' \wedge q' \xrightarrow{\delta_2} q'' \implies q \xrightarrow{\delta_1+\delta_2} q''$.

A2) *(Time continuity)*
$q \xrightarrow{\delta_1+\delta_2} q'' \implies \exists q'.(q \xrightarrow{\delta_1} q' \wedge q' \xrightarrow{\delta_2} q'')$.

A3) *(Time-determinism)*
$q \xrightarrow{\delta} q' \wedge q \xrightarrow{\delta} q'' \implies q' = q''$.

The concepts defined for LTS can be easily extended to TTS: path and trace contain also the time elapse as label, the simulation relation requires the same amount of time to be observed by the two TTS and two TTS that are composed need to strongly synchronize also on time elapse.

1.1.2 Timed (Input/Output) Automata

Describing complex RTES directly as transition systems is a cumbersome, and somewhat unreasonable, task due to the large number of states and actions they define. More compact notations are required to be used for modeling such systems. Alur et al. have introduced in [7] the *timed automata* concept for modeling timed systems which has now become the standard to describe RTES.

Definition 1.10 (Timed automaton). A *timed automaton* (TA) $\mathcal{A}$ is a 6-tuple (L, L^0, Clk, C, A, D) where:

- L is a finite set of locations.
- L^0 is an initial location.
- Clk is a finite set of clocks.
- $C : L \to \Phi(Clk)$ is a function that associates to each location some clock constraint. $\Phi(Clk)$ consists in individual constraints of type $x \leq c$, $c \leq x$, their negation or their conjunction, where x is a clock and c a constant in $\mathbb{Q}$.
- A is a set of observable actions.
- $D \subseteq L \times A \times \Phi(Clk) \times 2^{Clk} \times L$ is a set of transitions of type (s, a, ϕ, λ, s') where s and s' are locations, a the action to be performed, ϕ is the clock constraint which specifies when the transition is enabled (also called guard) and λ the set of clocks to be reset.

Basically, a timed automaton is a finite automaton enriched with a set of real-valued clocks that allow to measure time delays. In this computational model, time passes at the same rate for all clocks. Each location modeled in the automaton is mapped by a function C to a clock constraint that describes how much time can elapse before a transition is executed. Once a transition is enabled, i.e. the corresponding guard is satisfied by the current valuation of clocks, the discrete action may be performed and a set of clocks may be reset to 0. At any instant when a clock is read, it contains the amount of time elapsed since its last reset.

The semantics of a timed automaton is given by a timed transition system. A state of the TA becomes a pair $(s, v) \in L \times \mathbb{R}_+^{Clk}$ that consists of a discrete location s and the valuation of all clocks denoted by v. From a state (s, v) such that $v \vDash C(s)$ where $\vDash$ denotes satisfaction, the TA can progress either by a discrete transition (performing an action) or by letting time elapse. The transition relation $\longrightarrow$ of the corresponding TTS is the largest relation generated by the following rules:

1. for $a \in A$, $(s, v) \xrightarrow{a} (s', v')$ if $\exists(s, a, \phi, \lambda, s') \in D$ such that $v \vDash \phi$, $v' = v[\lambda]$ (i.e. clocks from λ are set to 0, while the others keep their values) and $v' \vDash C(s')$.
2. for $\delta \in \mathbb{R}_+$, $(s, v) \xrightarrow{\delta} (s, v')$ if $v' = v + \delta$ (i.e. clock values are augmented with δ) and $v, v' \vDash C(s)$.

Then, the corresponding TTS is the 4-tuple $(Q,\ \theta,\ A,\ \longrightarrow)$ where $Q = L \times \mathbb{R}_+^{Clk}$ and $\theta = (L^0, \mathbf{0})$ where $\mathbf{0}$ denotes the valuation of all clocks to 0.

The same notions as for TTS are used to describe the behavior of a TA, i.e. path and trace. Compound systems can be modeled as networks of timed automata which are composed by a parallel composition operator. Similarly to TTS, the composition operator requires the synchronization of common actions and time elapse and interleaving of the other actions.

A different notation for timed automata has been proposed in [108] which we will discuss later together with its input/output extension.

Due to their success for modeling real-time systems, several extended versions of TA have been proposed in order to better describe different aspects of the target systems. We may cite probabilistic TA [20], hybrid TA [100], TA with urgency [31, 32, 14] and Timed Input/Output Automata [108, 62]. We discuss in the following the latter two extensions which inspired our work.

Timed Automata with Urgency

In [32] is argued that modeling time progress conditions by clock constraints on both states and transitions is somewhat inconvenient, especially when hard time bounds with respect to performing actions have to be specified. Timed automata with urgency propose to describe time elapse only on transitions by shifting the clock constraints defined on states to the notion of *deadline* for transitions. A transition of the TA with urgency has the structure $(s,\ a,\ g,\ d,\ \lambda,\ s')$ where g and d are the guard, respectively the deadline, of the transition and consist in individual constraints of type $x\#c$, $x - y\#c$, their negation or conjunction, where

c is a constant, $x, y \in Clk$ and $\# \in \{\leq, <\}$. Transitions are correctly defined if $d \implies g$.

From the relative position of the deadline d with respect to the guard g we distinguish several *urgency* types for actions:

- *eager* when $d = g$. This condition specifies that time progress is disabled. If the automaton is in a state and such type of transition is enabled, the automaton cannot remain in the state and has to immediately execute one of the enabled transitions. Remark that the executed transition may be different from the transition that disabled time elapse.
- *lazy* when $d = false$. This condition specifies that time progress is enabled and unbounded. If the automaton is in a state and a lazy transition is enabled, the automaton may take the transition or may let time elapse. Time can progress to infinity if there is no stronger condition defined on the other outgoing transitions.
- *delayable* when d is the falling edge of a right-closed g. This condition specifies that time progress is enabled and bounded by a limit. If the automaton is in a state where a delayable transition is enabled, time can elapse up to the limit, when the transition becomes eager. Then an enabled transition must be immediately executed. A delayable transition cannot be disabled without forcing its execution.

At the TTS level, time can elapse in a state as long as the time progress condition $c_s = \bigwedge_{i \in I}(\neg d_i)$ holds, where I denotes the set of outgoing transitions from the state.

Timed Input/Output Automata

A second extension for TA consists in partitioning the set of actions into *inputs* and *outputs*, denoted *timed input/output automata* (TIOA). For an automaton, input actions correspond to the actions performed by the environment and are represented by the prefix ?, while output actions correspond to actions it executes and are represented by the prefix !. Therefore, for the automaton under study, inputs are uncontrollable and outputs are controllable.

In literature, several distinct notations have been defined for TIOA: [62, 61] build their specification framework on top of the standard definition for TA for which timed game semantics is considered during refinement, [108] defines a representation similar to hybrid TTS that satisfies several interesting compositionality results and [113] extends TA with urgency where refinement is given by a conformance relation.

TIOA of David et al. [62, 61]. As mentioned, [62, 61] define a game-based specification theory for timed systems. By specification theory we mean that besides the parallel composition and refinement operators, conjunction and quotient operators are defined. The conjunction allows computing the largest specification that refines two independent ones defined over the same language. The quotient operator computes from a partially implemented specification, the coarsest specification of the remaining (not implemented) part. Generally, the aim for a specification theory is to provide substitutivity results allowing for compositional design.

The input/output extension is defined for the Alur-Dill TA [7], as follows.

Definition 1.11 (Timed Input/Output Automaton [62, 61]). A *timed input/output automaton* $\mathcal{A}$ is a TA $(L,\ L^0,\ Clk,\ C,\ A,\ D)$ where $A = I \cup O$, I denoting the set of inputs and O the set of outputs.

The timed game semantics is expressed by the refinement relation, which in this framework consists in *alternating timed simulation* at the TTS level.

Definition 1.12 (Alternating timed simulation). Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two input/output TTS. $\mathcal{A}_1 \leqslant \mathcal{A}_2$ if and only if $\exists R \subseteq Q_1 \times Q_2$ a binary relation such that $\theta_1 R \theta_2$ and $\forall q_1 R q_2$:

- whenever $q_2 \xrightarrow{?a}_2 q_2'$ for some $q_2' \in Q_2$ then $\exists q_1' \in Q_1$ such that $q_1 \xrightarrow{?a}_1 q_1'$ and $q_1' R q_2'$,
- whenever $q_1 \xrightarrow{!a}_1 q_1'$ for some $q_1' \in Q_1$ then $\exists q_2' \in Q_2$ such that $q_2 \xrightarrow{!a}_2 q_2'$ and $q_1' R q_2'$,
- whenever $q_1 \xrightarrow{\delta}_1 q_1'$ for some $q_1' \in Q_1$ and $\delta \in \mathbb{R}_+$ then $\exists q_2' \in Q_2$ such that $q_2 \xrightarrow{\delta}_2 q_2'$ and $q_1' R q_2'$.

This framework is implemented in the ECDAR toolkit [64, 63] which provides means for compositional design of real-time systems.

A different specification theory for the same input model has been defined in [44], where refinement is defined by a timed trace inclusion relation. This specification framework is used for checking that safety and bounded liveness properties (see Section 1.2) are preserved at component substitution.

TIOA of Krichen et al. [113]. In [113], the refinement relation is similar to the trace inclusion relation. The timed input/output conformance relation (*tioco*) states that a concrete TIOA with urgency $\mathcal{A}_1$ conforms to an abstract TIOA with urgency $\mathcal{A}_2$ if and only if for any trace σ of $\mathcal{A}_2$, the set of enabled outputs and time delays of $\mathcal{A}_1$ after any run matching σ is included in the set of enabled outputs and time delays of $\mathcal{A}_2$. The main difference consists in the condition $A_2 \subseteq A_1$ over the set of labels, while trace inclusion requires for both automata to share the same set of labels. Therefore, $\mathcal{A}_1$ may accept more inputs than its abstraction and so performing a refinement of signature.

TIOA of Kaynar et al. [108]. The definition of timed input/output automata from [108] resembles the definition of TTS, but this framework is more general by allowing to describe hybrid systems. By hybrid systems we mean that the contained clocks may have multiple and therefore different rates.

A system is represented by a non-deterministic, possibly infinite-state, automaton. An automaton can own local variables and a set of clocks. The state of the automaton is given by the valuation of all its internal variables. The state of an automaton can evolve either due to a discrete transition or over a dense time interval by following a trajectory. Each discrete transition corresponds to an action. The set of actions is partitioned into internal actions which are not observable by the environment and external actions used to communicate with the environment divided into inputs and outputs. A trajectory is an increasing function over the time interval J, $J \subseteq \mathbb{R}_+$, with values in the set of states of the automaton. For an internal variable v, the trajectory describes its evolution over time elapse. We remark that the evolution of clocks is expressed by a differential equation in a trajectory, one for each clock, and therefore they may have different time rates.

Definition 1.13 (Timed Input/Output Automaton [108]). A *timed input/output automaton* $\mathcal{A}$ is a 9-tuple $(X, Clk, Q, \theta, I, O, H, D, \mathcal{T})$ where:

- X is a finite set of discrete variables and Clk is a finite set of clocks. We denote by $Y = X \cup Clk$ the set of internal variables.
- $Q \subseteq val(Y)$ is a set of states where $val(Y)$ is the set of valuations for Y. A valuation is a function defined on Y that associates to each variable a value from its domain.
- $\theta \in Q$ is the start state.
- I is a set of input actions and O a set of output actions. We denote by $E = I \cup O$ the set of external actions.
- H is a set of internal actions. We denote by $A = E \cup H$ the set of all executable actions.
- I, O and H are pairwise disjoint sets.
- $D \subseteq Q \times A \times Q$ is a set of discrete transitions.

- $\mathcal{T}$ is the set of trajectories. Each trajectory is a function $\tau : J_\tau \to Q$, where J_τ is a real interval of type $[0, t]$ or $[0, \infty)$ with $t \in \mathbb{R}_+$.

Notation. The same notation convention as for LTS holds here. For a trajectory τ we denote by $\tau.\mathit{fval} = \tau(0)$ and by $\tau.\mathit{ltime}$ the supremum of its domain. Then $\tau.\mathit{lval} = \tau(\tau.\mathit{ltime})$. The same annotation $\mathrm{x} \xrightarrow{\tau} \mathrm{x}'$ can be used where, $\forall \tau \in \mathcal{T}$, $\mathrm{x} = \tau.\mathit{fval}$ and $\mathrm{x}' = \tau.\mathit{lval}$.

A trajectory τ is *closed* if its domain is a closed interval. $\tau' = \tau\lceil[0, t]$ with $t \in J_\tau$ is called a *prefix* where $\lceil$ denotes the restriction operator. τ' is a *suffix* if $\exists t \in J_\tau$ such that $\tau' : [0, \tau.\mathit{ltime} - t] \to Q$ if τ is closed or $\tau' : [0, \infty) \to Q$ if τ is open, and $\tau'(u) = \tau(t + u)$, i.e. τ' obtained by restricting τ to $J_\tau \cap [t, \infty)$ and left-shifting it such that $J_{\tau'}$ starts in 0.

A timed input/output automaton has to satisfy the following axioms:

A0) *(Existence of point trajectories)*
$\forall \mathrm{x} \in Q, \gamma(\mathrm{x}) \in \mathcal{T}$ where $\gamma(\mathrm{x}) : [0, 0] \to \mathrm{x}$ maps 0 to x.

A1) *(Prefix closure)*
$\forall \tau \in \mathcal{T}$, $\forall \tau'$ a prefix of τ, $\tau' \in \mathcal{T}$.

A2) *(Suffix closure)*
$\forall \tau \in \mathcal{T}$, $\forall \tau'$ a suffix of τ, $\tau' \in \mathcal{T}$.

A3) *(Concatenation closure)*
Let $\tau_0\tau_1\tau_2\cdots$ be a (finite or countably infinite) sequence of trajectories in $\mathcal{T}$ such that, for each nonfinal index i, τ_i is closed and $\tau_i.\mathit{lval} = \tau_{i+1}.\mathit{fval}$. Then $\tau_0^\frown\tau_1^\frown\tau_2^\frown\cdots \in \mathcal{T}$, where $^\frown$ denotes the concatenation operator, i.e. the union between a first closed trajectory and a second one right-shifted such that its start time coincides to the limit of the first one.

A4) *(Input actions enabling)*
$\forall \mathrm{x} \in Q, \forall a \in I, \exists \mathrm{x}' \in Q$ such that $\mathrm{x} \xrightarrow{a} \mathrm{x}'$.

A5) *(Time-passage enabling)*
$\forall \mathrm{x} \in Q, \exists \tau \in \mathcal{T}$ such that $\tau(0) = \mathrm{x}$ and either
 1. $\tau.\mathit{ltime} = \infty$, or
 2. τ is closed and some $a \in H \cup O$ is enabled in $\tau.\mathit{lval}$.

An important feature of this framework is its expressiveness. The urgency stereotypes presented above can be easily described by the set of available trajectories in a state, as it is showed in the following:

- *eager* will give place in a state only to the point trajectory;
- *lazy* is translated as the set of all possible trajectories. Let $2^{[\mathbb{R}_+^0]} = \{[0, t] | t \in \mathbb{R}_+\} \cup \{[0, \infty)\}$, i.e. the set of intervals defined in $\mathbb{R}_+$ having their left-limit

equal to 0. Then the set of trajectories available in a state is $\{\tau_{I_\tau} | \forall I_\tau \in 2^{[\mathbb{R}_+^0]}, \tau_{I_\tau} : I_\tau \to Q\}$

- *delayable* until d gives the set of possible trajectories on the domain $\{[0,t) | t \leq d, t \in \mathbb{R}_+\}$.

The behavior of the automaton is described by the same notions as for TTS, the path (or execution) and the trace, while for the trace actions in H are abstracted. The set of traces of the automaton $traces_{\mathcal{A}}$ can present two properties: closure under limits and closure under time-extension. *Closure under limits* informally means that any infinite sequence whose prefixes are traces is also a trace. *Closure under time-extension* denotes that any trace can be extended with an open interval trajectory. The formal definitions are presented in [108].

Two automata can be composed if they are compatible: they do not share any variables or internal actions and outputs are disjoint. Syntactically, the parallel composition operator models the input/output synchronization and interleaving of all other unmatched actions, while matching actions become output actions.

Definition 1.14 (Parallel composition). If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two compatible TIOAs then their *composition* $\mathcal{A}_1 \parallel \mathcal{A}_2$ is defined to be the tuple (X, Clk, Q, θ, I, O, H, D, $\mathcal{T}$) where:

- $X = X_1 \cup X_2$ and $Clk = Clk_1 \cup Clk_2$.
- $Q = \{\mathrm{x}_1 \cup \mathrm{x}_2 | \mathrm{x}_1 \in Q_1, \mathrm{x}_2 \in Q_2\}$. Note that $\mathrm{x}_1 \cup \mathrm{x}_2$, which denotes the set union of functions x_1 and x_2, is well defined since the domains of x_1 and x_2 are disjoint.
- $\theta = \theta_1 \cup \theta_2$.
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$.
- $O = O_1 \cup O_2$.
- $H = H_1 \cup H_2$.
- D is the set of discrete transitions where for each $\mathrm{x} = \mathrm{x}_1 \cup \mathrm{x}_2$, $\mathrm{x}' = \mathrm{x}'_1 \cup \mathrm{x}'_2 \in Q$ and each $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ if and only if for $i \in \{1,2\}$, either
  1. $a \in A_i$ and $\mathrm{x}_i \xrightarrow{a}_i \mathrm{x}'_i$, or
  2. $a \notin A_i$ and $\mathrm{x}_i = \mathrm{x}'_i$.
- $\tau \in \mathcal{T} \Leftrightarrow \tau \lceil X_i \in \mathcal{T}_i$, $i \in \{1,2\}$.

This theory defines several refinement relations: trace inclusion also named implementation, forward and backward simulation, history and prophecy relations. In the following we consider trace inclusion to verify refinement between automata and we present two compositionality results from [108] that are proved to hold and which we will later exploit in our contribution.

Theorem 1.1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two TIOAs with the same external set of actions such that $\mathcal{A}_1 \preceq \mathcal{A}_2$ and $\mathcal{E}$ a TIOA compatible with both $\mathcal{A}_1$ and $\mathcal{A}_2$. Then $\mathcal{A}_1 \parallel \mathcal{E} \preceq \mathcal{A}_2 \parallel \mathcal{E}$.*

Theorem 1.2. *Let $\mathcal{K}$, $\mathcal{E}$, $\mathcal{G}$ and $\mathcal{A}$ be TIOAs such that $\mathcal{K}$ and $\mathcal{G}$, and $\mathcal{E}$ and $\mathcal{A}$ have the same external set of actions and each of $\mathcal{K}$ and $\mathcal{G}$ are compatible with $\mathcal{E}$ and $\mathcal{A}$. If*

- *$traces_{\mathcal{A}}$ and $traces_{\mathcal{G}}$ are closed under limits and under time-extension,*
- $\mathcal{K} \parallel \mathcal{A} \preceq \mathcal{G} \parallel \mathcal{A}$
- $\mathcal{G} \parallel \mathcal{E} \preceq \mathcal{G} \parallel \mathcal{A}$

then $\mathcal{K} \parallel \mathcal{E} \preceq \mathcal{G} \parallel \mathcal{A}$.

Theorem 1.1 shows that trace inclusion is compositional, i.e. refinement is preserved by composition. Theorem 1.2 is an *assume-guarantee* style of substitutivity result which allows to replace abstract safety properties with implementations if the refinement relation is satisfied in the abstract context of the other. The conditions of closure under limits and time-extension model that both $\mathcal{A}$ and $\mathcal{G}$ are safety properties with arbitrary time-passage. The safety aspect is given by the closure under limits characterization of the set of traces: if all traces in a chain of successive extensions "satisfy" a property, then so does its chain limit. And, if a trace "satisfies" a property, then so do all its prefixes.

1.1.3 Interface Theories

Another compact notation used to model a system is the *interface automaton*. Introduced in [65, 66, 117], it is defined as an automata-based language that captures the order in which methods of a system are called (inputs) and the order in which the system calls external methods (outputs). It proposes a fixed model of computation by strongly synchronizing inputs and outputs at composition under an optimistic approach: the order of methods called by the environment is correct. This condition represents the main difference with respect to I/O automata: while an I/O automaton has its inputs enabled in any state, the interface automaton enables input reception in a precise order, i.e. interface automata model certain assumptions about the environment in which they are used. The semantics of an interface automaton is given by an ATS, the refinement relation being defined as alternating simulation which is preserved under composition. The set of actions of two interface automata involved in a refinement relation must satisfy a covariant relation on inputs (i.e. more inputs enabled in the concrete automaton) and a contravariant relation on outputs (i.e. less outputs enabled in the concrete automaton).

In consequence, interface automata are aimed to answer the following composability question: is there an environment in which components can work together correctly? More generally, a component from an interface theory can be used as a common specification by different teams working concurrently on partial designs while ensuring independent implementability.

Different versions of interface theories have been defined in the literature: timed interfaces [67], modal interfaces [38, 27] or interfaces for synchronous communications [153, 81].

Interface theories have also been subject to the definition of specification theories. In [39], an interface automaton is extended with an explicit representation for inconsistent states, while its semantics is given by an input/output labeled transition system. The formalism defines parallel composition, refinement based on trace inclusion, conjunction and quotient at the semantical level. Specification theories have been considered also for modal interfaces in [26, 145, 27].

1.1.4 Summary

Our aim is to formally model reliable RTES designed with UML/SysML. Therefore, the framework needs to explicitly describe the following features: time progress via clocks, input/output distinction of the performed actions and input-enabledness for modeling asynchronous communications. Based on this characterization, we consider that the most appropriate semantical model to describe such system designs is the input/output TTS, with an operational representation as TIOA.

There are two important mathematical representations for TIOA, which we compare in Table 1.1 with respect to several key elements needed in UML/SysML modeling like describing internal computation steps, providing means to incorporate asynchronous communication and already available compositional reasoning results.

Describing the semantics of an RTES in the TIOA theory defined in [62, 61] suffers from several limitations. First of all, internal complex computational steps can not be described in this formalism since the silent action ε is not member of the set of actions partitioned only into inputs and outputs. Moreover, extending the notation to include silent actions is not a solution since the refinement relation does not take into account such transitions. Actually, refinement is defined as strong timed alternating simulation at the semantic level (game-based treatment) and is compositional. The second limitation is related to the parallel composition operator which defines a synchronous communication. Since the TIOA does not

	TIOA of [62, 61]	TIOA of [108]
Modeling internal computation	✗	✓
Asynchronous communication	✗	✓
Compositional refinement	✓	✓
Specification algebra	✓	✗

Table 1.1 – Comparison of TIOA representations.

define the silent action ε, the consumption of a received message, which is specific to asynchronous communication, cannot be described. A more general remark is that this framework defines a specification theory suited for compositional design based on the added operators conjunction and quotient, where quotient is however a partial specification, i.e. it cannot always be computed.

On the other hand, the theory from [108] presents several features that make it more suitable to describe the semantics of RTES. The definition of a TIOA includes a set of internal variables and actions besides clocks, which allow to model more complex computations. The definition of trajectory allows the modeling of hybrid systems and it includes the timed semantics of the TA from [7]. The equivalent timed semantics can be obtained by restricting trajectories to constant values for internal variables and the derivative describing clock rates to 1. Moreover, trajectories can be easily used to describe the urgency notation of TA, which makes the framework's expressiveness richer.

The parallel composition operator also describes synchronous communications but can be simply adapted to asynchronous one: a predefined variable *queue* can store inputs — the automaton is input-enabled — which are handled later by internal actions. Several refinement relations are available, from which the framework proves two interesting compositional properties for trace inclusion: refinement is a preorder (i.e. reflexive and transitive) compositional relation and can be used in an *assume/guarantee* reasoning style.

Based on these features, we chose to represent the semantics of our RTES in the framework of [108] since it is well defined and the compositionality results it proves are a prerequisite of our contribution.

1.2 Verification Techniques for Formal Models

In order to certify that the system model is correct, we employ *formal methods* for guaranteeing the satisfaction of system requirements. A particular type of requirements is represented by timed properties: they describe the accepted actions a system may perform and when they may occur, e.g. a response should be delivered in x time units. In this thesis we concentrate on a non-exhaustive category of requirements related to timed behavior, which are described in Section 1.2.1.

Several validation and verification techniques can be used during the development phases to check if the system meets its specified requirements. Validation methods are aimed to confirm if the right product has been designed, i.e. if the obtained result is correct. For example, *tests* allow to find bugs in a system design either during implementation or at component integration by specifying a set of inputs and inspecting if the obtained result is the required one. *Interactive simulation* is also a validation technique in which the execution of the system is supervised by the user in order to detect possible requirement violations. Their main limitation is that they are non-exhaustive, i.e. only a partial set of behaviors can be considered, and, therefore, the absence of property violations does not imply the satisfaction of the requirement.

In contrast, verification techniques cover all behaviors a system can exhibit for verifying requirement satisfaction. By definition, they aim to inspect if the system has been correctly developed. There are two main approaches to explore the system's entire behavior: *theorem proving* and *model-checking*. Theorem proving is an automated method in which the system and the requirement are specified in some mathematical formalism, while interactive mathematical proofs are realized on the formalization using logic calculus. The main inconvenience is that they necessitate the interaction with a user that will guide the proof. In consequence, this technique is hard to master by non-experts. Instead, model-checking is an automatic method directly applicable on the formal model of a system and which hides the implementation/proof details from the user. Therefore, it has gained a lot of popularity in both academical and industrial world. Yet, it suffers from some limitations in the verification of very large systems or systems-of-systems. Under these considerations, we chose to use in the following model-checking, described in Section 1.2.2, for proving requirement satisfaction and for which we are led to define an improvement for its limits.

1.2.1 Overview on System Requirements

We distinguish several types of requirements that a reactive system may have to satisfy which are summarized hereafter.

Reachability. This property models that a certain state of the system can or cannot be reached during execution. Reachability is often used to characterize safety properties and bounded liveness ones.

Safety. Such a property verifies the absence of catastrophic consequences under given conditions, i.e. something bad never happens, by describing the set of acceptable behaviors for the system. Safety is usually formalized as a reachability property where bad states must be avoided at execution. If a safety property is not satisfied, a finite execution of the system is sufficient to find a counterexample.

Liveness. Informally, a liveness requirement states that something good eventually happens. Therefore, liveness describes *progress* conditions about the system. There are two possible causes for not achieving progress: (1) a deadlock situation in which two components cannot execute discrete transitions since they wait for each other to finish their computation and (2) a livelock situation in which two components continuously change their state with regard to one another but without finishing their computation. In order to detect inconsistent behaviors, infinite time is needed for observing the system: the action *eventually* occurs sometime in the future. A particular case is represented by the *bounded liveness* property which states that the property must be satisfied in a maximal delay, i.e. an action occurs within a given period. Bounded liveness requirements can be described by timed safety properties.

Reliability. It describes the system's ability to behave continuously correct. Reliability is depicted by fault-tolerance properties which are relevant for safety-critical systems, i.e. if certain faults are produced by the environment the system's behavior is consistent.

Performance. Such property characterizes the speed of operation/computation for a system. An interesting sub-class is represented by the response time requirements which describe how fast an output is produced by the system based on the received inputs. We consider that the timed behavior of a system requirement can be interpreted in this manner, with a particular consideration when refining requirements into components.

1.2.2 Model-Checking

Model-checking [141, 49, 52] is one of the most well-known fully automated verification techniques that provides an answer to the requirement satisfaction problem. It consists in exploring all the behaviors of a system and ascertaining that the property holds for all states and all orders of events. A system is usually given by its timed transition systems, while a property can be expressed by different formalisms like Linear Temporal Logic [138], Timed Linear Temporal Logic [102], Computational Tree Logic (CTL) [49], Timed CTL [6] or timed property automata [4].

The latter consists in modeling the requirement that the system $\mathcal{S}$ must satisfy with a timed automaton $\mathcal{O}$ such that $A_{\mathcal{O}} \subseteq A_{\mathcal{S}}$. Then the algorithm proceeds by synchronizing $\mathcal{S}$ with $\mathcal{O}$ and the property checking sums up to some reachability problem of a bad state (i.e. the system does not satisfy the property) or a good state (i.e. the system satisfies the property) for $\mathcal{S} \parallel \mathcal{O}$, i.e. *reachability analysis*. Therefore, model-checking is tailored for verifying (timed) safety and liveness properties.

Being an enumerative method of the state space, model-checking guarantees that all behaviors of the system have been explored and therefore the obtained result is correct. A second advantage of model-checking is that it can provide a counterexample when the property is not satisfied. This counterexample corresponds to one or more problematic behaviors that should be considered for correcting the model.

However, this characterization is also the cause of the main drawbacks for model-checking. First of all, explicit-state model-checking can be considered only for systems that have a finite number of states. Secondly, when verifying industrial-grade models, the state space's size quickly becomes combinatorial, which makes practical model-checking impossible. This problem is commonly known in literature as the *state space explosion* problem.

In order to overcome it, several optimization techniques have been proposed. The main goal is to make model-checking scalable such that more complex systems can be subject to formal verification.

Reduction techniques. The aim of these techniques is to reduce the state space either by using a symbolic representation for states [103] or applying partial order reduction [86]. A symbolic representation allows to have more compact and abstract notations like zones or difference bound matrices to group time elapse or binary decision diagrams to structure sets of states. Partial order allows to explore a representative subset of system runs by keeping only one independent sequence of actions and eliminating the others. Two actions a_1 and a_2 are independent if in any state both sequences a_1a_2 and a_2a_1 are enabled and they reach the same target state. Therefore, depending on the degree of independence, a large number of transitions may be eliminated, but this still may not be sufficient to be able to explore the rest of the state space.

Compositional model-checking. Since a system is usually built by assembling several components, another approach is to exploit the architectural structure for verifying requirement satisfaction, denoted compositional reasoning [68, 53, 94]. Given a system $\mathcal{S} = \mathcal{S}_1 \parallel \mathcal{S}_2 \parallel \ldots \parallel \mathcal{S}_n$ and φ the requirement to verify, the basic idea is to decompose φ on components and to individually verify that each component $\mathcal{S}_i$ satisfies its corresponding φ_i. Since components are in general smaller, the state space is reduced and therefore, the verification process becomes tractable. Two remarks need to be made about this approach: (1) the decomposition of requirements is a laborious task since there is no methodological guideline available and (2) components can rarely satisfy a property on their own without taking into consideration the behavior of the environment. Dependencies between components and their environment need to be clearly identified and verified, exercise that is sometimes tedious because such dependencies are frequently mutual.

Abstraction. This technique [51, 69] is somewhat the dual of compositional model-checking by working on components rather than requirements. For each component $\mathcal{S}_i$ an abstraction $\mathcal{S}'_i$ is provided and the satisfaction of the requirement φ must be proved on the composition $\mathcal{S}'_1 \parallel \mathcal{S}'_2 \parallel \ldots \parallel \mathcal{S}'_n$. The first remark we make about this approach is that the abstraction has to be correct: $\mathcal{S}_i$ refines $\mathcal{S}'_i$ and $\mathcal{S}'_i$ preserves all the properties of $\mathcal{S}_i$ needed for proving the satisfaction of the global requirement. By applying model-checking for the satisfaction of the global requirement on the abstract model, a counterexample is generated in case of an error. Then, one needs to inspect the counterexample on the concrete system in order to decide if it is valid, i.e. the concrete system contains an incorrect behavior, or *spurious*, i.e. the error is introduced by the modeled abstraction and it does not hold on the concrete system. In case of a spurious error the concerned $\mathcal{S}'_i$ must be refined, while in case of a valid error the component $\mathcal{S}_i$, which violates φ, must be refined. These steps are summarized by the counterexample guided abstraction refinement (CEGAR) [50, 29, 1, 97] method, which may be applied on $\mathcal{S}'_i$ in order to guarantee its correctness. Secondly, the same remark as for compositional reasoning with respect to the dependency upon the environment holds here, since the used refinement relation may not take into account the environment's behavior.

1.2.3 Summary

Model-checking has gained real success in the verification of formal timed system models. This can be claimed by the large number of available implementations like UPPAAL [119] for the TIOA of [62, 61], CADP [79], nuSMV [45], Spin [105], IF toolset [34] for TIOA with urgency and extended with data structures, etc.

The main drawback of model-checking is that it suffers from the combinatorial state space explosion problem which makes it inappropriate for large system's monolithic verification. In order to alleviate this problem two interesting optimizations have been developed which still present some limitations, i.e. abstraction and compositional reasoning. Our contribution is situated in the field of compositional reasoning and abstraction modeling and refinement by introducing an intermediate level of *contracts* for specifying requirements. Therefore, a contract will model with a *guarantee* an abstract behavior for a component given that the environment behaves in a specified way while it will also play the role of a partial specification for a requirement. The notion of guarantee represents at the same time for a component an abstraction but also a constraint given by the requirement's decomposition on components.

In order to take into account the environment during refinement, the contract defines an *assumption* over its behavior from the point of view of the component to which it is related. In consequence, the guarantee is a correct abstraction for the component in the context of the assumption, while the assumption is a correct abstraction of the environment only in the context of guarantee. This type of reasoning is called *circular* and is merely a particular case of the abstraction optimization by restricting the refinement relation to an abstract context (instead of all possible behaviors).

Then, instead of generating in one step the entire state space, model-checking will be applied only locally on the system's decomposition: each contract needs to be satisfied either by the component to which is related or, if it is defined for a composed component, by a set of contracts which refines it. Indeed, depending on the hierarchical structure of the system, the approach can be iterated for a composed component downwards to its sub-components, which can be either atomic or composed depending on their complexity and the desired level of granularity. We assume, at this level of detail, that the requirement is expressed as the starting guarantee.

The number of refinement relations that need to be verified is linear in the number of components the system models. We remark here that for two identical components which satisfy the same contract, refinement verification needs to be performed only once. This fact underlines the reusability feature of the method.

In consequence, this method which we call *contract-based model-checking* can effectively tackle the state space explosion problem, provided that the defined abstractions are small enough.

1.3 Conclusion

This chapter focuses on the correct development of RTES with respect to the desired requirements, which requires the use of formal models and methods. In consequence, we have outlined several modeling frameworks, both operational and semantical, that could describe RTES modeled with UML/SysML. Since UML/SysML designs allow to describe components having timed behaviors with complex internal computational steps and a reactive model of computation, we chose to formalize the system design with the Timed Input/Output Automata of [108], while their semantical representation is given by timed input/output transition systems and the refinement relation between components as timed trace inclusion.

As verification technique, we focus on model-checking since it is a fully automated method currently used in industrial practice that provides a yes/no answer to the requirement satisfaction problem. Yet, model-checking suffers from the combinatorial state space explosion problem, which makes the method intractable for very large systems or systems-of-systems. We have briefly introduced the context of our work, denoted *contract-based model-checking*, which proposes an optimization to the monolithic approach by applying model-checking only locally on components, i.e. it combines the abstraction and compositional optimization approaches. Therefore, the notion of contract is introduced in the system design in order to specify for each component an abstract and partial behavior with respect to the running requirement, as well as the abstract context in which this constraint needs to hold. Hence, model-checking will be applied several times, linear in the number of components, but we can assume that the abstractions are small enough in order to efficiently alleviate the state space explosion problem.

2 High-Level Modeling Languages and Associated Environments for Real-Time Embedded Systems

Formal models allow to precisely specify the system to develop and its requirements based on a clear and sound syntax and semantics. Furthermore, formal specifications can provide a correct design and implementation of systems due to the verification and validation techniques they are subject to. However, despite their strong theoretical bases, employing formal languages during development is a difficult task for non-experts and an important source of errors given the abstract nature inherent to this type of language.

With the advent of object-oriented design and analysis, the Unified Modeling Language (UML) [91] has been proposed and standardized as common means to generally specify computer systems. UML and lately Systems Modeling Language (SysML) [90], a UML extension specialized to systems and requirements engineering, provide a standard graphical notation for the visualization of the system design which is accompanied by a semi-formal specification. By semi-formal specification we mean that UML/SysML define a well-formed syntax, while their semantics is sometimes ambiguous or unspecified (i.e. open variation points). Therefore, they define an intuitive set of concepts for modeling systems halfway between natural language and formal models, but they cannot be exploited as defined for the validation and verification of system designs. Several environments have been developed in order to allow the verification of a system design by transforming it into a formal specification, which constitutes the input of a model-checking tool.

A different notation than UML is the Architecture Analysis & Design Language (AADL) [148], a standard specialized in the architectural modeling of embedded systems. The aim of AADL is to provide means to model distributed fault-tolerant architectures with support for the analysis and validation of system requirements. Yet, it does not contain any mechanism for explicitly specifying timed behavior.

The Specification and Description Language (SDL) [106, 149] is a standard proposed for the development of real-time event-driven asynchronously communicating systems that involve parallel computations, primarily applied for telecommunication systems. Different from UML/SysML, this notation presents mathematical rigor via its clear and consistent formal semantics, which allows to generate implementation code that can be further used for validation and verification, e.g. simulation. But also, UML is more general than SDL which can be considered a domain specific language due to its notions originating from programming languages: a UML profile [107] has been designed to represent SDL notions and enable UML model editors for SDL [112, 111].

In this chapter we present a non-exhaustive state of the art in modeling and verifying system designs with UML/SysML. Section 2.1 generally presents UML/SysML and their extensions, called profiles, that cover the real-time aspect. Section 2.2 lists a set of verification and validation environments for semi-formal system designs.

2.1 UML/SysML and Related Profiles for Real-Time Systems

Modeling in UML/SysML consists in designing different diagrams that cover multiple views of a system, namely: requirements, architecture and behavior. The requirement diagram, specific to SysML, describes the system's properties and the relations between them, where a property is expressed in natural language. The architecture of the system is given in UML/SysML in class diagrams/block definition diagrams by classes/blocks used to describe types and in composite structures/internal block diagrams by objects/block instances that are assembled together and communicate via interaction points and links. Finally, the behavior of the system is expressed either by state machine diagrams, sequence diagrams or activity diagrams. A state machine is a notation similar to finite-state automata that can execute and invoke actions, as well as changing the state of the owning component. A sequence diagram describes the collaboration between components by modeling the execution order of events. An activity diagram describes the workflow of a component with possible communication between components.

The real-time features of a system are partially covered in UML 2.x. In order to fully describe RTES, the following aspects need to be offered by high-level modeling languages: models for physical time, timing specifications and facilities, and models for physical resources and concurrency. Yet, UML 2.x defines modeling elements for concurrency (e.g. active objects, composite states for an object) and two time-related data types — Time and TimeExpression — that can be used to express timing constraints in sequence/state/timing diagrams.

SysML partially inherits real-time notions from UML 2.x: the concurrency aspect is present in SysML, yet the timed behavior of a component cannot be modeled. Moreover, SysML includes notions that allow modeling synchronous distributed fault-tolerant architectures (i.e. time-triggered).

Previous versions of UML, namely 1.x, did not cover any time-related aspect. In consequence, the Schedulability, Performance and Time Specification (SPT) [89] profile has been defined in order to enable the modeling of quality of service, resource, time and concurrency, and performing predictive schedulability and performance analysis. The profile owns several time-related stereotypes, e.g. Clock, TimeValue and TimeInterval, which can be directly applied by the user on the corresponding modeled concepts. In consequence, there is no formal semantics specified for these stereotypes, which makes the verification and validation task cumbersome.

The SPT profile, available only for UML 1.4, has been replaced in 2009 by the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) Profile [129], compliant with UML 2.x. MARTE is a more generic profile, which provides several computational paradigms (asynchronous, synchronous and time-triggered) and facilitates model-specific analysis (e.g. performance, schedulability) by annotating models with the essential information. Similar to SPT, MARTE defines a set of time-related stereotypes for which it does not specify a clear formal semantics.

SPT has served as source for other profiles targeting real-time specifications. We mention here the OMEGA Profile [87], which defines a clear and coherent formal semantics for a subset of UML and two extensions: concepts for modeling timing constraints and formalization of requirements with observers. OMEGA has a proprietary tool that allows to perform formal validation and verification with the IFx toolset[1]; prior to this thesis the toolset was compliant only to UML 1.3. We have recently updated the profile to make it comply with UML 2.x and SysML in [125, 126, 58, 57]: the architecture of a system is described by block definition diagrams and internal block diagrams, while its behavior is given by state machine diagrams. The new features are considered for implementation in the IFx2 Toolset[2].

The Timed UML and RT-LOTOS Environment (TURTLE) [12] is a UML 1.5

[1] http://www-if.imag.fr/

[2] http://www.irit.fr/ifx/

Profile that defines notions for concurrency and timing constraints. It introduces several composition operators, which have to be explicitly described, for modeling concurrency: parallel composition, synchronization, sequence, invocation and preemption. Temporal operators include deterministic and non-deterministic delays. Timing constraints can be modeled in the activity diagram for each object. TURTLE has also been updated recently, which led to the definition of a SysML Profile extended with time named AVATAR [137]. The behavior of a system is given in an AVATAR design mainly by state machines extended with the temporal operators defined in TURTLE, namely delays of behavior suspension and computation time for instructions. Requirements are formalized using the Temporal Expression Property LanguagE (TEPE) [110], while both profiles are supported by TTool[3].

The MADES language [140] proposes to use a subset of SysML and MARTE to describe RTES with a focus on the hardware specification and allocating resources to hardware, while avoiding incompatibilities from the usage of both profiles. Clock specifications are based on the MARTE's Time Modeling concepts, therefore the corresponding stereotypes are instantiated on each case study instead of defining general applicable clock types. The Zot tool [15] enables the verification of MADES designs via a transformation to temporal logic formulae.

Other approaches include the UML-RT profile [150] that allows modeling complex event-driven and possibly distributed real-time systems. This profile defines concurrency notions, yet it does not support time and timing constraints modeling. A more detailed discussion about UML-related real-time profiles can be found in [82, 115].

2.2 Verification Tools for System Designs

The verification and validation of a system design demands the transformation of the semi-formal model into a formal model, as well as specifying requirements with the aid of a formal language. Within the past years, several tools have been developed to support essentially the model-checking of system designs modeled with UML/SysML/MARTE, but also with other component languages derived from UML. Such tools, based on already developed model-checkers, are either directly integrated into modeling tools or they provide independent transformation from the input model (usually expressed in an XMI [93] file) to the target language

[3] http://ttool.telecom-paristech.fr/

of the considered model-checker. For example, SysML Companion[4] provides as extensions model-checking in UPPAAL and Spin.

IBM Rhapsody[5] includes validation by simulation in its toolbox. In [59], a system model developed with Rhapsody where the behavior is expressed by state machines is subject to model checking in UPPAAL. The system's properties, even though they are initially described in requirements diagrams, have to be expressed by the user as CTL formulas on the obtained timed automata network model.

HUGO/RT[6] translates a system model where the behavior is described by time annotated state machines for the UPPAAL model checker [109]. The requirement to verify is modeled by the user as a time annotated sequence diagram which is transformed into a timed property automaton.

ARTISAN Real-Time Studio[7] system designs are transformed in [5] into input models for the nuSMV model-checker. The framework translates the behavior described in sequence, activity and state machine diagrams into the Configuration Transition Systems intermediate format which is model-checked with respect to the automatically generated properties as well as manually specified ones as CTL formulas.

The Topcased Environment[8] provides connections with UPPAAL and CADP for SysML models. The AGATE project [147] translates SysML activity diagrams into timed automata in order to analyze the execution of the system on a mono platform processor, where time descriptions are based on MARTE stereotypes. The FIACRE language [25] is proposed as pivot representation in Topcased to connect AADL, SDL, UML and SysML models to CADP or TINA[9]. TINA is a model-checker for Timed Petri Nets [139], a formalism used for representing distributed timed systems. However, the automated transformation from UML/SysML to FIACRE is not yet available.

The academic toolbox TTool[10] transforms AVATAR models [137] into UPPAAL. Safety requirements are expressed using the Temporal Expression Property LanguagE (TEPE) [110] that uses the parametric diagrams of SysML to express logical and temporal relations between block attributes and signals. The system design

[4] http://www.realtimeatwork.com/software/sysml-companion/

[5] http://www-03.ibm.com/software/products/en/ratirhapfami

[6] http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

[7] http://www.atego.com/products/atego-modeler/

[8] http://polarsys.org/

[9] http://projects.laas.fr/tina/

[10] http://ttool.telecom-paristech.fr/

is transformed into a network of timed automata, while the requirements become timed property automata.

The IFx toolset[11] transforms an OMEGA design (modeled with IBM Rhapsody or Papyrus[12]) into asynchronously communicating timed automata with urgency. Timed safety requirements are formalized by observers, a special type of objects that monitor the executed events within the system, which are then translated into timed property automata. The tools available in the IF Toolset[13] can be applied on the obtained formal specification: interactive simulation of the OMEGA model and model-checking combined with reduction techniques.

Lately, MARTE designs have also been considered as inputs models for model-checking. In [152] mode behaviors (i.e. a specialization of state machines) are transformed in timed automata for the UPPAAL model-checker and a requirement is expressed using the Clock Constraint Specification Language (CCSL) [10] which allows to specify causality constraints, chronological and timed properties. In [80] activity diagrams are transformed in Timed Petri Nets, the input model for TINA, while several categories of requirements proposed by the framework are available for verification, like best/worst-case response/execution/traversal time, schedulability and synchronization-related properties.

2.3 Conclusion

Semi-formal modeling languages have been rapidly adopted by the industrial development of RTES for the high-level specification of systems and their requirements due to their graphical, compact and easy to use notation. For example, Airbus Defence and Space (ADS)[14] has deployed SysML for capturing the system requirements of the new version of Ariane-5 launcher. Yet, high-level models often contain bugs mainly due to the ambiguous semantics these languages define which leads to different interpretations of the design by system engineering teams, but also due to the insufficient knowledge of the formalisms and jargon used by system and software engineering teams during development. Ensuring that the system design satisfies its requirements from the first development phases is a major concern since it permits to implement safe systems having a greater quality and with reduced production costs.

[11] http://www.irit.fr/ifx

[12] http://www.eclipse.org/papyrus/

[13] http://www-if.imag.fr/

[14] http://airbusdefenceandspace.com/

	Time-related concepts	Formal semantics	Tool support
UML 2.x	✓ (concurrency and timing constraints — data types)	✗	✓[15]
SysML	✓ (concurrency)	✗	✓
SPT/MARTE	✓ (timing constraints, shedulability, performance — stereotypes)	✓ (Timed Automata, Timed Petri Nets[16])	✓
UML-RT	✗	✗	✓ (Rational)
OMEGA	✓ (concurrency, timing constraints, performance)	✓ (Timed Automata with urgency)	✓ (any modeler and IFx Toolset)
TURTLE/AVATAR	✓ (delays, composition operator)	✓ (RT-LOTOS, Timed Automata)	✓ (TTool)

Table 2.1 – Comparison of UML/SysML and related profiles for modeling RTES.

Therefore, the validation and verification of semi-formal system models has become a crucial task at each design step. Model-checking is widely used via different tools for guaranteeing the satisfaction of requirements. Yet, the limitations that were expressed at the formal level hold also at the semi-formal level. In consequence, the improvement we discussed in Section 1.3 should be transferred to high-level system designs.

Table 2.1 presents a summary of the modeling frameworks previously described and their associated validation and verification environments. We can remark that the standards do not impose any constraint with respect to the formal semantics of the concepts they define, and that a formal semantics is rather associated with the toolset used for verification. The OMEGA and AVATAR approaches are better candidates for rigorous system engineering due to their inherent formal aspect, which is a great aid in modeling unambiguous components that serve as common specifications for the different teams implementing the system. The main inconvenience for AVATAR is that only TTool can be used to design such systems, whereas OMEGA is compatible to any modeling tool offering the option to add profiles to a model and to export the model in an XMI file, making it more compelling.

[15] Several tools are available both for modeling such designs, as well as their verification.

[16] The formal semantics depends on the toolset used for validation and verification, since the standards do not describe one.

3 Contract-based Reasoning for Hierarchical Systems of Components

Contract-based design is an emerging development paradigm for the modular and compositional design and verification of systems, which has its roots in the axiomatic representation of programs [104, 2] and has been fruitfully applied in object-oriented software engineering [121].

The principle is to define a partial and abstract specification in the form of a contract that every component must implement. The specification is partial since it has to be defined only with respect to the running requirement and abstract since it does not provide implementation details of the requirements. In consequence, a specification represents the projection of a requirement on the component that has to satisfy it. With respect to the system design, contracts are a valuable asset since they allow for: component substitutivity and reuse, incremental development and independent implementability. Complemented with a formal approach, the design process can result in correct-by-construction implementations. Besides design, contracts can be used for: mapping and tracing requirement to components, tracking requirements during their evolution, correct virtual integration of components [60] and, most importantly, compositional verification.

Recent work has explored the notion of contract in order to define a recipe for developing systems in the form of meta-theories, but also to directly provide theories and tools for specific component formalisms. Recall that by meta-theory we mean a generic framework, independent of a particular component specification: in order to obtain a working theory, one has to formalize the component specification and the refinement relations contract/components are subject to, while proving possibly required compositionality results. However, the application of the contract-based (meta-)theories in high-level modeling languages has been left aside. Until now the concept of contract as it is used in systems engineering mainly refers to the couple pre/post condition for operations or for model transformations.

In this chapter, we present an overview of the contract-based approaches as they are defined and used in both fields. Section 3.1 describes the formal contract-based reasoning (meta-)theories defined in the literature and their implementations, for which we review the basic notions. Section 3.2 depicts the types of contracts that were proposed for high-level modeling languages and their aim.

3.1 Contract-based Meta-Theories and their Implementations

The interest for examining meta-theories comes from their formal generic aspect and the sound reasoning method they define. The generic aspect allows to instantiate the meta-theory for any component language and therefore obtain a working framework that could immediately be applied for contract-based reasoning in a particular context. A meta-theory's reasoning is based on a set of compositional properties that allow to correctly establish requirement satisfaction. In consequence, the instance contract-based framework will also guarantee the soundness of its application, provided that these properties are proved in the meta-theory's instantiation.

3.1.1 A Meta-theory for Contract-based Reasoning

In the following we describe the meta-theory proposed in [143, 144, 142] which constitutes the basis of our work. We provide the key notions and properties that allow to compositionally reason with contracts for hierarchical systems. An important asset of this meta-theory is the methodology with which it is equipped, as illustrated in Figure 3.1, and which is explained below.

Assume, at any level of the hierarchical decomposition of a system, a subsystem S obtained from the composition of several components K_1, K_2,. . ., K_n for which we want to prove that it satisfies a requirement φ. This meta-theory leaves open the choice of the composition operator in order to accommodate various operators. In order to simplify the presentation, we will assume the existence of a notion of component compatibility, to be defined in the instance of the meta-theory, and the existence of a composition operator for every pair of compatible components, denoted $\|$, which is unique and associative. Thus, S is obtained from the composition $K_1 \parallel K_2 \parallel \ldots \parallel K_n$.

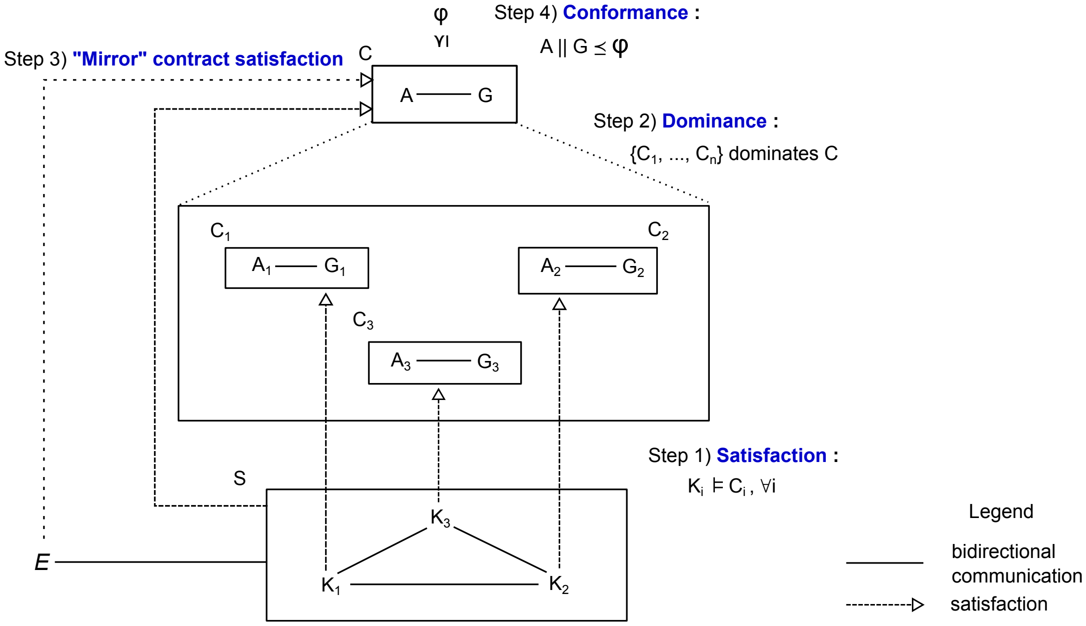


Figure 3.1 – Contract-based reasoning for a three-component subsystem ([142]).

In order to use the methodology, one starts by modeling a *contract* for each component the subsystem contains.

Definition 3.1 (Contract). A *contract* $\mathcal{C}$ consists of a pair of compatible components (A, G) where the component A is called the *assumption* and the component G is called the *guarantee*.

The contract $\mathcal{C}_i = (A_i, G_i)$ for the component K_i is an *abstract* model of how the component K_i contributes towards the satisfaction of the requirement φ. The assumption models the behavior of the environment of K_i and the guarantee models the expected behavior of K_i provided that the environment obeys the assumption. Note that different contracts can be expressed for one component, in particular when one wants to prove different properties on the subsystem S. The example in Figure 3.1 presents a subsystem S formed by three components K_1, K_2 and K_3 and its environment E with which it communicates. For each of these components a contract $\mathcal{C}_i$ is modeled.

In order to prove the satisfaction of φ, the reasoning proceeds by first proving that each component *satisfies* its contract, denoted $K_i \models \mathcal{C}_i$. To define *satisfaction*, the meta-theory relies upon the existence of a *refinement under context* relation. This relation between two components K_i and K_j in an environment E, denoted $K_i \sqsubseteq_E K_j$, informally means that the component K_i, when composed with the environment E, “behaves like” K_j when composed with the same environment. Although the choice of the refinement under context relation is left open, the meta-theory requires the operator to satisfy certain important properties, such as compositionality and correctness of circular reasoning; they are needed for proving Theorem 3.1 below.

Based on refinement under context, contract satisfaction is defined as follows:

Definition 3.2 (Contract satisfaction). Let $\mathcal{C} = (A, G)$ be a contract for a component K. Then K satisfies the contract $\mathcal{C}$, denoted $K \models \mathcal{C}$, if and only if $K \sqsubseteq_A G$.

Note that the meta-theory does not impose any constraint with respect to the signature of the components K, A and G. As we will later see, we allow A and G to concentrate only on a subset of a component's signature and on a part of its behavior. This provides the ability to keep a contract abstract, with only the essential information for the running requirement.

The second step of the reasoning shown in Figure 3.1 consists in defining a contract $\mathcal{C} = (A, G)$ for the entire subsystem S and proving that the set of contracts $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *dominates* the contract $\mathcal{C}$.

Definition 3.3 (Contract dominance). A set of contracts $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *dominates* a contract $\mathcal{C}$ if and only if for any set of components $\{K_i\}_{i=\overline{1,n}}$ the following holds: $K_i \models \mathcal{C}_i, i = \overline{1,n} \implies K_1 \parallel K_2 \parallel \ldots \parallel K_n \models \mathcal{C}$

The dominance relation as defined in this meta-theory involves component composition while avoiding the definition of a contract composition operator. Still, when using contract-based reasoning for verification we are not interested in manipulating implementations in order to establish dominance which are the main cause for combinatorial explosion in large systems. The meta-theory provides an alternative result which boils dominance down to a set of satisfaction checks, provided that two compositionality conditions hold in the framework's instance:

1. Refinement under context is compositional: $K_1 \sqsubseteq_{E_1 \parallel E_2} K_2 \implies K_1 \parallel E_1 \sqsubseteq_{E_2} K_2 \parallel E_1$. This property allows for incremental design by successively incorporating parts of the environment in the components under study, while refinement under context holds.
2. Circular reasoning is sound: $K \sqsubseteq_A G \wedge E \sqsubseteq_G A \implies K \sqsubseteq_E G$. This property allows for independent implementability by breaking down the dependency between the components and their environment.

These two compositionality results allow the meta-theory's reasoning to be sound.

Theorem 3.1 (Sufficient condition for dominance). *If compositionality and circular reasoning are sound, then for establishing that* $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *dominates* $\mathcal{C}$ *it is sufficient to prove that*

$$\begin{cases} G_1 \parallel ... \parallel G_n \models \mathcal{C}, \text{and} \\ A \parallel G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n \models \mathcal{C}_i^{-1}, \ \forall i \in \overline{1,n} \end{cases}$$

where $\mathcal{C}_i^{-1} = (G_i, A_i)$ *denotes the* "mirror" contract *of* $\mathcal{C}_i$*.*

The first relation requires the refinement of a more abstract guarantee by a set of more specific guarantees, while the second expresses that individual assumptions need to be refined by the other components' guarantees together with the overall assumption. Since real-life systems often exhibit a multi-layer architecture, the dominance step can be iterated upwards until reaching the contract of the component for which the property of interest φ is defined. To simplify the presentation, Figure 3.1 only shows one step of dominance.

In order to prove the satisfaction of the requirement φ, we have to make sure that the assumption A we made over the environment E and which is used in the top-level contract $\mathcal{C}$ is correct. The third step of the methodology consists in verifying the satisfaction of the "mirror" contract $\mathcal{C}^{-1}$. We note that this is necessary when the system under study S is an open subsystem. Yet, if the requirement φ is expressed on a closed system then there is no assumption to be defined and this step may be skipped.

The last step of the reasoning consists in proving that the top contract *conforms* to the requirement φ. The relation to be verified at this step is $A \parallel G \preceq \varphi$, where $\preceq$ denotes a property conformance operator, left open in the meta-theory of [142, 143].

The strategy detailed above is presented from a bottom-up point of view and assumes that the system has already been conceived and thus the designer has to deal with contracts for the satisfaction of the requirement. However, this methodology can also be applied in a top-down manner for the correct design of systems given the requirements it must satisfy. In a top-down approach, one starts by modeling the requirement φ the system under study must verify. Next, a top contract is defined that must conform to φ, where the guarantee is merely the requirement modeled as a component that interacts with an abstract environment. Next the contract is projected on a subset of contracts that must dominate it, given that the corresponding subset of components has been envisioned. Again this step can be iterated for a multi-layer architecture. For each contract, a (weakest) component is derived that conforms to the guarantee in the context of the assumption. A component can incorporate several contracts if the system needs to satisfy several requirements. Finally, a correct implementation is designed for the system's under study environment E that has to satisfy the "mirror" global contract, which can be realized independently of the component designs.

The reasoning can be easily applied to hierarchical systems with an arbitrary number of components. A generalization to recursively defined finite systems is provided in [142] by a grammar and a set of rewriting rules corresponding to the methodology.

This meta-theory has already been instantiated for several formalisms: Labeled Transition Systems with priorities [142] and data [99], Modal Transition Systems [143], BIP components [16, 88] and Heterogeneous Rich Components [22, 23, 24]. Yet, to the best of our knowledge, there is no documented application for timed systems with asynchronous communication.

Then, the following steps need to be taken in order to define a contract-based framework for a concrete component model — system designs with SysML in our case — by relying on this meta-theory:

1. Formally define the component framework — component, parallel composition and possibly refinement if different from conformance —, the conformance relation $\preceq$ and the refinement under context relation $\sqsubseteq_E$.
2. Prove that both *conformance* and *refinement under context* relations are preorders.
3. Prove that, for closed systems, conformance can be deduced from refinement under context, i.e. if $K_1 \sqsubseteq_E K_2$ then $K_1 \parallel E \preceq K_2 \parallel E$.
4. Prove that *refinement under context is compositional.* Compositionality is a prerequisite of Theorem 3.1.
5. Prove that *circular reasoning is sound.* This property is also a prerequisite of Theorem 3.1.

3.1.2 Related Contract-based Approaches

There are two lines of reasoning with contracts defined in the literature: *circular reasoning* which requires for both refinements in the hypothesis to hold in their abstract corresponding context and *assume/guarantee reasoning* (AG) which consists in strengthening the hypothesis and requiring for the assumption of the contract to be refined by the environment regardless of how the component behaves. Formally, with the previous notation the rule is written: $K \sqsubseteq_A G$ and $E \sqsubseteq A$ then $K \sqsubseteq_E G$. In contrast, the second hypothesis is written $E \sqsubseteq_G A$ for circular reasoning.

Most of the related work relies on AG reasoning. Yet, AG is harder to achieve for systems like real-time embedded systems since they have mutual dependencies between components: it demands to find a strategy for breaking the symmetry of the dependency between a component and its environment by finding a component which can guarantee its property independently of its environment, task that may be very complex for our target systems. This equates to the same limitation of the abstraction technique for model-checking. Therefore, we consider that sound circular reasoning is more interesting and sufficient in a contract-based approach to allow for independent implementability of components.

Related contract-based meta-theories have been mainly developed for *specification theories*. We recall that by a specification theory we mean a complete algebra that contains parallel composition, logical composition, quotient and refinement operators. The aim for a specification theory is to provide substitutivity results allowing for *compositional design*, while its usage in the *compositional verification* of requirement satisfaction did not receive as much attention. In contrast, the meta-theory described in Section 3.1.1 provides the minimal set of operators needed for both design and verification, even though it may show up an overhead especially during design: for example, it is up to the designer to model the weakest component that must satisfy several contracts. Logical conjunction [19] and quotient can be formalized and added to our framework, whereas their use should be independent from the reasoning described above that is sufficient and sound for verification.

The meta-theory of [17] is built for a specification theory and is similar in many points with [142]. The main differences concern (1) the specification of contracts and (2) the method for reasoning with contracts. Regarding the first item, the framework of [17] does not support signature refinement, i.e. the ability of a contract to concentrate only on some of the inputs/outputs of the component while abstracting away the others, which is allowed in [142] and, as we will later see, explicitly used in our framework instance. A partial solution is presented in [18] where contracts are defined on a subset of the component's signature via ports. However, [18] does not allow to reason individually on contracts: at each design step, a specification consists in a component and the complete set of contracts on ports such that the union of their signatures is equal to the signature of the component. In consequence, composition and refinement of specifications involve all the elements they define, i.e. components and ports. Moreover, signature refinement between specifications is not allowed since the refinement operator requires for all elements from specifications to have the same signature.

Regarding the method for reasoning with contracts, both meta-theories from [17, 18] subscribe to the AG reasoning: they use the assume-guarantee rule that requires to abstract the environment of a component regardless of how the component abstractly behaves. For proving dominance, the meta-theory of [17] heavily relies on a (derived) *contract composition* operator, which can be used to compute the "strongest" contract $\mathcal{C}_1 \boxtimes \mathcal{C}_2$ satisfied by the composition of two components that respectively satisfy $\mathcal{C}_1$ and $\mathcal{C}_2$. The dominance step is reduced to the following proof obligation: $\mathcal{C}_1 \boxtimes \mathcal{C}_2 \boxtimes \ldots \boxtimes \mathcal{C}_n \preceq \mathcal{C}$, where $\mathcal{C}$ is the dominated contract. However, contract composition is partial, i.e. it can be *undefined* for certain pairs of contracts, since it is based on the quotient operator which is itself partial. The meta-theory of [18] also uses a specification composition operator which in this case requires for the contained components and port contracts to be pairwise composed. So, the use of such operators may turn out difficult for large and complex systems. Moreover, in both meta-theories, the failure of proving dominance is not explicitly discussed and it is not clear how the user could identify the cause and mitigate it. In contrast, the method from [142] allows to infer, based on the proof obligations that constitute the sufficient conditions for dominance, which contract is faulty. On a more general note, neither the meta-theory of [17] nor [18] explicitly describe how a system requirement has to be formalized and how its satisfaction can be achieved via contracts. We consider that the reasoning methodology, illustrated on an example in Figure 3.1, of the meta-theory from [142] is an important asset in its application to concrete domains.

The meta-theory of [17] has been applied in [63] for the TIOA specification theory of [61, 62] and implemented in the ECDAR toolset [64], where the timed gamed semantics of refinement can be verified [37, 33]. However, several aspects of this theory make it impractical for representing the semantics of timed components described in SysML or UML, as it is mentioned in Section 1.1.4. The same meta-theory has been applied for communicating components modeled as a set of traces in [47, 48] and is implemented in the OCRA tool [46]. In this instantiation, a contract is given by a pair of hybrid LTL assertions on the set of traces and dominance is verified with a sufficient condition similar to the one define in [142] even though circular reasoning is not explicitly demanded.

Assume-guarantee reasoning is a long-standing line of research, although classical approaches deal with logical specifications [53, 95, 2]. The more recent approach of

[42, 43] deals with specifications in the form of sets of I/O traces for verifying both safety and progress requirements characterized by finite traces and handles the signature refinement in contract satisfaction although only for *untimed* specifications. Working directly on traces without an intermediate operational specification is difficult due to the complexity of completely describing the set of traces a component executes, separated into accepting ones and failing ones in order to model a safe behavior. We mention that in this case input/output automata are considered for graphical representation, but their formalization is not presented. Moreover, the reasoning approach is similar with the one proposed in [17], in particular with regard to contract composition. As such, the remarks made above with respect to the reasoning methodology remain valid for [42].

Interface theories [65, 66, 67, 117, 39], formally described in Section 1.1.3, are also related to contracts since they can be used to express assumptions and guarantees for a component, albeit they do it in the same specification. An interface encompasses the assumption represented by its inputs and the guarantee represented by its outputs and describes how a component and its environment are expected to interact. Some of the specification theories developed in this context are based on variants of TIOA close to the one that we are using, like in the case of [44]. The theory of [44] allows reasoning for both safety and bounded-liveness properties on finite timed traces. Generally, the notion of a contract that merges the assumption and the guarantee is well suited to derive compatible environments in which components can work together since it models how a component should behave. In fact, an interface plays the same role as the composition between individually modeled assumption and guarantee, which has to be manually computed by the designer. Keeping assumptions and guarantees separate has several advantages: (1) it allows to model component properties as stand-alone automata and (2) it allows for the refinement of the assumption or guarantee as well as the component to be performed independently.

3.2 Contracts in High-Level Modeling Languages

As described in [28], software engineering distinguishes 4 classes of contracts: *syntactical* ones that describe the types a component can handle, *behavioral* ones that add constraints about the use of a component and which are inspired from the Eiffel programming language [121], *synchronization* ones that specify the global behavior and interaction of components and *quality-of-service* ones that can quantify the expected behavior of components, e.g. response delay.

Plentiful work has focused on syntactic and behavioral contracts in order to provide a mechanism for solving the *composability* problem. In [156], the authors make the distinction between an output contract which is offered by the component and an input contract on which the component relies, where a contract is a generic view of an interface. Then contracts serve as type specifications for components during the development phases of a system. In [114], contracts are defined as OCL [92] pre/post conditions for operations, which are validated through simulation. In [36, 35], the same type of OCL contracts are expressed for all modeling elements in order to perform model transformations or to model the execution semantics of UML notions, where the latter can be seen as a particular case of model transformation.

The Kmelia component model [120, 11] provides means to verify the functional correctness of behavioral contracts for services, as well as compatibility issues between components. The meta-model defines contracts for operations or interfaces and models explicitly the verification results. With respect to behavioral contracts, one is modeled for an operation as pre/post condition pair, while the behavior of the operation is modeled as an extended Labeled Transition System. Formal verification of contract satisfaction is realized by transformation of the component language to formalisms such as CADP or AtelierB[1].

Synchronization contracts have been considered in [136] where on each component boundary an interface is defined with attributes, operations and a protocol state machine that describes the response of the component to sequences of events constraining their order. A contract is defined on connectors between two component boundaries, where protocol state machines are used to verify the compatibility of components. Despite the instantiation of these notions in SysML via the stereotype mechanism, the theory is not supported by a formal framework which can provide answers about the satisfaction of a contract.

In [13] the notion of contract is used as a mean to formalize requirements which are verified on system-of-systems expressed as Stochastic State Transition Systems. A contract is modeled by an OCL expression which may contain Contract Specification Language patterns [151] that correspond to Bounded Linear Temporal Logic operators. Then model-checking is directly applied in order to prove the satisfaction of the requirement. The notion of contract merely equates to the formalization of requirements for which a compositional reasoning approach — for the design of systems and their verification — is not provided.

Contracts have also been considered for synchronous SysML and AADL architec-

[1] http://www.atelierb.eu/

tures in [157, 55, 78], which propose a reasoning similar to the one presented in Section 3.1.1. A contract is defined as a pre/post condition pair on components with respect to their inputs/outputs, while each condition is modeled by a past-time Linear Temporal Logic formula. Informally, contract satisfaction models that if the assumption holds at any previous moments, then the guarantee will also hold at the current moment. The theory provides a mechanism to verify dominance directly on contracts similar to Theorem 3.1: iterative verification of the satisfaction of individual assumptions by the other guarantees on which it depends and the global assumption and verification of the global guarantee by the global assumption and individual guarantees. However, circular reasoning is required to hold only locally at one moment in time: components are allowed to refer to guarantees of the others in earlier instants in time such that at one particular moment there is no circularity in the model. This is supported by the synchronous communication of components with one-step communication delay. Top-level requirements are modeled in the Property Specification Language, but their satisfaction by a contract is not formalized. Furthermore, this theory does not define a syntax for contracts in SysML or AADL.

3.3 Conclusion

Despite the obvious advantages a contract-based development approach offers, system engineering has not yet adopted it to design modular component-based system models or to perform compositional verification of requirement satisfaction. This is mainly due to the difficulty of designing correct contracts, as well as providing the good compositionality results that ensure the correctness of the reasoning. Indeed, designing contracts that can be refined towards correct implementations is a source of overhead since methodological guidelines were not yet available. However, by comparing it with the expense of developing systems on which model-checking fails to provide a result and where errors are discovered late and by very costly processes, we can assume that the entailed overhead is less important.

Our statement is based on the recent research that has focused on developing recipes for correct contract-based reasoning, which we denoted by meta-theories. Moreover, the meta-theory described in Section 3.1.1 is equipped with a rather extended methodological guideline for applying contract-based reasoning, which aims also to relieve the overhead for modeling contracts by providing some design rules. Therefore, we consider that contract-based design is an interesting approach for developing correct systems and also for alleviating the limits model-checking presents.

Our contribution consists in grafting contract-based reasoning in the model-driven design and requirement verification process for real-time embedded systems described with SysML. Therefore, in this chapter we inspected the types of contracts that were proposed for UML/SysML designs. A majority of contract modeling concepts tackle the syntactical aspect — they describe the types (i.e. requests) a component can handle — and the behavioral aspect — adding usage constraints on types or components. These two categories of contracts merely enforce the statical compatibility of components. The behavior of a component is considered by synchronization contracts that describe the desired order of interaction. The definition of contract given in the meta-theory from Section 3.1.1, falls under this category since the assumption/guarantee explicitly describe (timed) ordered behavior for a component and its environment. Synchronization contracts are used in the Kmelia component language — a textual representation for web services — but only at the component's operations level, which are modeled with Labeled Transition Systems, thus untimed specifications. In the framework of [136], synchronization contracts are modeled on connectors for checking the compatibility of linked components, but without providing a formal theory. In the following we provide a definition of synchronization contracts in SysML that covers the whole dynamics of components, which is to the best of our knowledge the first to address this aspect.

In order to support the correct reasoning with contracts, we have presented a set of meta-theories, which need only to be instantiated in order to obtain a working framework — in our case for the SysML component model extended with time. We chose to instantiate the meta-theory of [143, 144, 142] due to its thoroughness and soundness, compact and clear notation where a minimal number of operators have to be defined in order to work and, finally, the application guidelines which are a tremendous advantage for systems engineering development processes.

Part II

Modeling and Reasoning with Contracts in SysML

4 The SysML Context

SysML is a industrially used standard, extending a subset of UML, that allows modeling the architecture of a system, its behavior and its requirements. Yet, the standard is quite rich and defines different points of view of the system that are not all needed for real-time embedded systems (RTES) design. Moreover, our aim is to have a representation similar to the hierarchical component-based system illustrated in Figure 3.1. Therefore, we identify a subset of SysML modeling elements for designing RTES and which will constitute our component language. Two extensions need to be considered for modeling such systems since SysML does not offer the corresponding aspects: time and timing constraints modeling, and formalization of requirements. Indeed, SysML is a semi-formal component language in which requirements are expressed in natural language. Providing a clear and coherent syntax and semantics for the considered modeling elements makes the system design suitable for formal verification. We describe such rules with the OMEGA Profile, which has been adapted to SysML and enforces rigorous system engineering.

Section 4.1 presents the set of notions from UML/SysML we use for modeling RTES. In Section 4.2 we introduce our working context, the OMEGA Profile. Finally, Section 4.3 describes the running example, a simplified version of an Automated Teller Machine, on which we will illustrate the notions defined in this thesis.

4.1 A SysML Subset for Modeling Asynchronous Component-based Systems

Recall that modeling in UML/SysML consists in designing different diagrams that cover multiple views of a system, namely: architecture, behavior and requirements, where the latter are specific to SysML. In the following we present a subset of the modeling elements we use for RTES as defined in UML in order to keep the description simple. The application to SysML is straightforward by using the counterpart concepts defined in the *UML4SysML* package of the standard [90].

Architectural modeling elements

A component-based system is structurally modeled in UML by two notions: the class which allows to define types that are to be used in the model and the composite structure that allows to cope with the growing complexity of large systems by describing how class instances (generally called *components*) are composed and interconnected in a hierarchical structure.

A class can model the following features: properties (attributes and parts) and relations (associations, compositions and generalizations). Besides classes, a UML model can contain other structured types, like signals and interfaces. Signals represent the most important notion for communication between components. It can own parameters, while their receptions are structured into interfaces. Interfaces are especially used in our component model as typing mechanism for components on interaction points.

Components are assembled together in composite structures that describe their communication via interaction points and links: an interaction point is represented by a port and the (sender, receiver) pair of components are hooked up by a connector between corresponding ports. Therefore, components are deemed to exchange asynchronous signals only via ports and connectors. As we will later see, making components communicate only via ports help ensure the rigorous typing of the model.

The requests which are offered/required by the component at its interaction points are described within interfaces that are required to type ports. For the sake of clarity, we demand ports to be typed with only one interface, which either describes the allowed inbound requests if it is declared as provided or the allowed outbound requests if it is defined as required. Yet, we remark that typing ports with only one interface makes them unidirectional and therefore certain typing inconsistencies that may appear when transforming the model into an implementation language are avoided [126]. Moreover, we observe that the aim of such a port is to provide interactions with respect to one functionality. We do not consider such modeling cumbersome; indeed, industrial designs taught us that having a unique type for ports is sufficient and that, sometimes, duplicating ports that originally were bidirectional brings more clarity to the model. The SysML standard introduces flowports to model synchronous communications. However, since we focus on asynchronous message passing, we forbid the modeling of flowports and we make use only of standard ports.

The default behavior of a port is to transfer requests from one side to another depending on its direction, inbound or outbound. Behavior ports are used to specify that the requests arriving at the port are directed to or from the behavior of the component owning the port.

Behavioral modeling elements

The behavior of a system is expressed in UML by several diagrams: state machines, sequence diagrams or activity diagrams. We select in the following to use state machine diagrams due to their notation similar to timed automata.

Therefore, the behavior of system components is expressed by state machines, possibly involving detailed action descriptions which can be further structured into operations. As previously mentioned, operations can be modeled by two signals: one call signal from the source which will wait for the operation completion and one return signal which models the end of the computation and which can have return values if they are required.

To describe actions, we use a subset of the action language formalized by the fUML standard [130]. This language covers the following notions: signal output, variable assignment and expression valuation. Signal output which in SysML is also called send action can be sent to a port if and only if the port type knows how to handle it, i.e. a signal reception is defined within the port type.

The state machine of a class describes its behavior in terms of finite states and transitions. The behavior moves from one state configuration to another by transitions, which can define a guard, a trigger and a, possibly structured, effect. The guard and the trigger model conditions that need to be evaluated to true for the transition to be enabled. The effect describes the computations that the component realizes. Again, for the sake of clarity, we consider that the transition's elements are evaluated in the order they are modeled: the guard, followed by the trigger and next, each effect in the order they are modeled.

Requirements in SysML

The requirement diagram, which is specific to SysML, describes the system's properties and the relations between them. The main inconvenience is that a property is expressed in natural language, which makes it unsuitable for formal validation and verification. Several formalisms are available to represent requirements, e.g. temporal logics, automata-based languages, etc., and implemented in different tools. In Section 4.2 we describe the instantiation with *observers* that is available in our working context. However, since we are interested to define our contribution in the general SysML context, we denote requirements in this thesis by the meta-class *SafetyProperty*.

4.2 Real-Time and Requirement Formalization: the OMEGA Profile[1]

OMEGA UML/SysML is an executable profile dedicated to the formal design and validation of real-time embedded systems and integrated in the IFx Toolbox[2] [34] which provides means to simulate and model-check such models. The aim of the profile is to define a precise and coherent operational semantics adapted to formal timed analysis techniques by providing a set of well formedness rules on the UML subset presented in Section 4.1 and two extensions for modeling timed behavior and formalizing timed safety requirements. The OMEGA-IFx approach has been applied on several industrial-grade case studies [127, 128] for which it proved its efficiency, but, as we will later see for the case studies presented in this thesis, the verification method is still not sufficiently robust.

The choice of OMEGA as working context is also motivated by the usage of the OMEGA-IFx approach in industrial practice, as illustrated by the Full Model-Driven Development for On-Board Software process [58, 57]. The interest of this approach is highlighted in [58, 57], the criteria considering the smooth usage with respect to both modeling elements and formal semantics and being more reliable and less costly compared to a more traditional one relying on informal paper documents and tests for discovering potential errors.

[1] This section is based on the following papers: [125], [70], [126], [124], [58], [57].

[2] http://www-if.imag.fr/

Well-formedness rules for strong typing

An important point left open by the UML standard is the correct definition of connectors. We distinguish two types of connectors in UML: delegation connectors from a component to its outward container and assembly connectors between components at the same architectural level. In order for connectors to be correctly used by traveling signals, OMEGA imposes the following limits with respect to the ports they link and their direction: a delegation connector between two provided ports is traveled from the environment to the inner component; a delegation connector between two required ports is traveled from the inner component to the environment and an assembly connector may be defined and traveled only from a required port to a provided one.

Moreover, the two categories of connectors can be defined only if the ports types coincide. This rule ensures that the receiving component knows how to handle the requests traveling through the connector. In consequence, OMEGA defines a dynamical typing of connectors based on the provided/required types at its ends via the concept of transported interface(s). The set of signals that can travel through a connector is given by the signal receptions defined within the ports type.

UML models a unicast communication: for each signal exchange there is one sender and one receiver. Yet, if two connectors are originating in the same port (i.e. the port is the source of the message), then one is chosen non-deterministically, which models an ambiguous semantics. OMEGA forbids such situations by demanding that in every port there is at most only one outgoing connector. Multicast communications (i.e. one sender multiple receivers) have to be explicitly modeled, as demanded by the UML standard.

This set of rules, as well as the typing restrictions we made over the concept of ports in Section 4.1, are formalized with OCL [92] in [126, 70] and can be statically evaluated on any system model using an OCL interpreter like Topcased. These rules ensure rigorous system engineering.

Extensions: timed behavior and observers

Real-time systems can be designed in OMEGA based on an extension the profile offers, inspired from Timed Automata with urgency [31]. One can cover prescriptive time modeling via the predefined concept of Timer, but also via time stereotypes on outgoing transitions that control how long time can elapse in a state.

The type Timer allows to measure durations and can be set to a relative deadline and reset by the two defined functions set() and reset(). Then timers (or clocks) can be easily used to describe timed transition guards. Upon deadline different actions may be executed like signal output, variable assignment, etc. We remark that the Timer can be also considered an instantiation of the *clockType* stereotype from MARTE with a dense time base, which owns two operations getTime/setTime.

Within a state machine, the timed behavior can also be controlled by urgency stereotypes on transitions such that one can specify more flexibly how logical time progresses. OMEGA introduces three stereotypes corresponding to the ones defined in Section 1.1.2:

- «*eager*» models that the transition has to be executed as soon as it is enabled, i.e. time elapse is disabled,
- «*delayable*» models that the transition has to be executed before a deadline d, i.e. time elapse is enabled and bounded, and
- «*lazy*» models that the transition can be executed at any moment in time or never, i.e. time elapse is enabled and unbounded.

Note that the *delayable* stereotype is derived from *eager* and *lazy* as follows: a timer is set with the deadline value before entering the state and there are two outgoing transitions from the state, one being typed lazy, while the other is eager and has a timeout guard for the deadline value. Therefore, in order to not overload the component model, we prefer to use the explicit modeling in the following.

We remark that the urgency stereotypes lack in MARTE, while they allow for a more compact modeling in OMEGA and prepare a formal timed semantics. In contrast, OMEGA proposes only one computational model, whereas the generic component modeling of MARTE is richer, but also generic with respect to the underlying operational semantics. The properties of simplicity and well-defined semantics are in our view complementary to what is aimed by MARTE.

The second extension of OMEGA concerns the formalization of dynamic safety timed requirements with *observers*. An observer is a special object that monitors the system events and gives a verdict about the (non-)satisfaction of the requirement it formalizes. It is modeled by a class stereotyped «observer» which has a local memory and a state machine to describe the requirement's behavior. In order to model the divergence from a nominal behavior scenario, observer states may be marked with the «error» stereotype when the action sequence has to be considered erroneous. Then at verification, an execution reaching the «error» state violates the satisfaction of the requirement.

In order to monitor the system, a set of probe events has been defined for observers. We present only a subset of such transition triggers that are used within this thesis:

- *send* monitors the output of a signal, and
- *acceptsignal* monitors the handling of a signal by the target component.

The trigger of an observer transition is defined by a *match* clause specifying the type of the event, the qualified signal name and the observer attributes that may receive related information like the signal parameter values. The effect of a transition solely relies on the attributes the observer defines, e.g. variable assignment.

4.3 The sATM Running Example

The following example will be used throughout this thesis in order to illustrate our contract-based approach and notions for system designs. We model a simplified Automated Teller Machine (sATM) for the withdrawal transaction only. The architecture of the system is represented in Figure 4.1 and consists of the following blocks: the *sATM* that contains the *CardUnit* responsible for the insertion and removal actions of a card and the *Controller* responsible for the withdrawal transaction, and the *User*. The model has been realized with the IBM SysML Rhapsody tool[3]. Note that the User models the environment of the sATM and, therefore, is not part of the system under study. We consider here only one of the possible behaviors a real customer can exhibit as it is described below.

The considered use case is the following: a customer is required to insert a card into the card unit of an *atm*. Then the *atm* will verify the amount available on the card and will propose several amounts within the accepted range to the user for withdrawal. The *atm* interacts with the customer via a console and can handle only one user at a time. The customer chooses an amount and waits for the *atm* to execute the transaction. The *atm* will display a message and eject the card. If the card is removed within 5 time units after being ejected, the amount will be distributed and the total amount available on the card will be updated. Otherwise, the card is retained and the *atm* becomes unavailable for further customers. The behavior is modeled in Figure 4.2 by a state machine for each component, where all transitions are eager.

We are interested in showing with the contract-based approach that the current model satisfies the following requirement:

[3] http://www-03.ibm.com/software/products/en/ratirhapfami

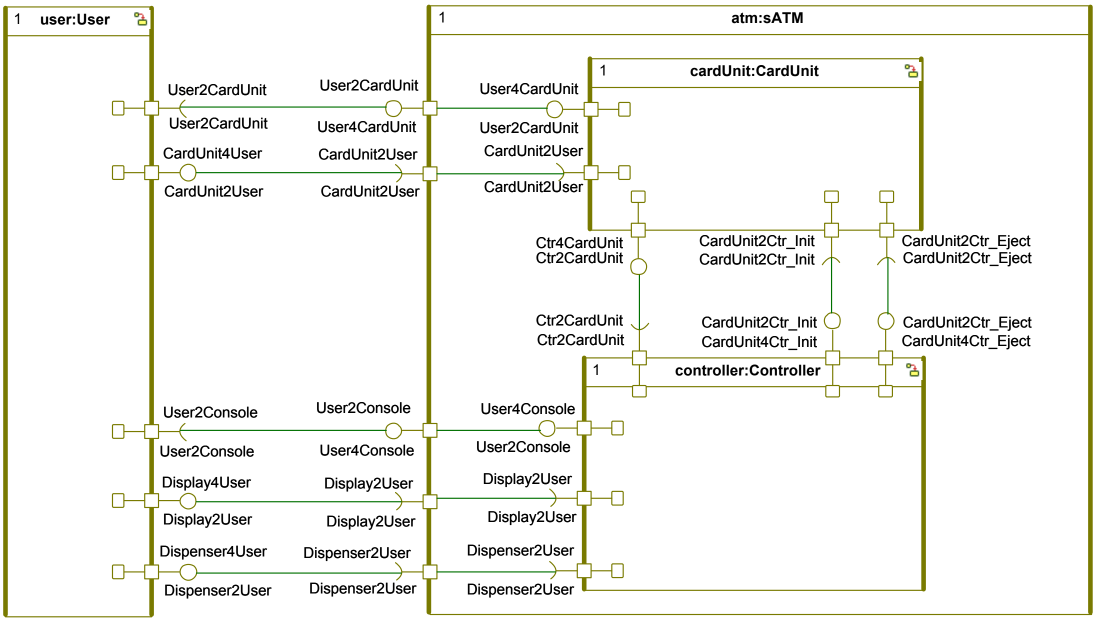


Figure 4.1 – Running example: the architecture of the simplified Automated Teller Machine (sATM)[4].

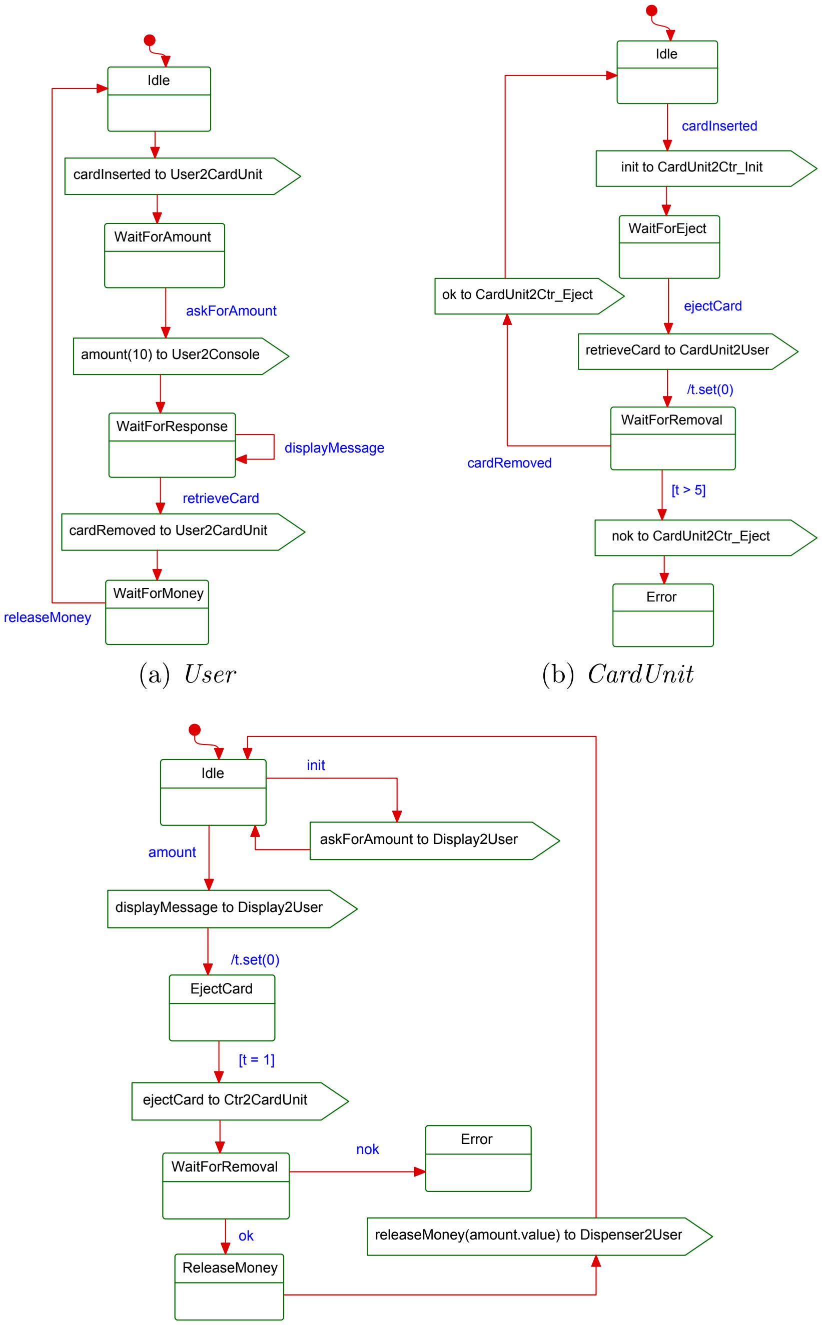
Requirement 4.1. *If the card is removed within 5 seconds after being ejected, the amount released by the sATM is the amount demanded by the customer.*

This requirement is formalized with an *observer* in Figure 4.3 from the point of view of the sATM. Initially, the observer waits in the state *Idle* for the customer to insert a card and select an *amount*. Next, it expects for the customer to remove the card from its slot once the latter is ejected by the machine: the *retrieveCard* output signal followed by the *cardRemoved* input signal sequence. Then, the sATM executes the *releaseMoney* operation. The values of the demanded and released amounts are modeled by a parameter of their corresponding signals. If the values coincide then the requirement is satisfied, otherwise the *Error* state is reached and the property is violated.

4.4 Conclusion

In this chapter we have presented a sufficient subset of SysML for modeling hierarchical component-based systems. To summarize, we have selected to represent

[4] *Notation.* The name of the ports starts with the name of the emitting component, followed by "2" if the port is modeled for the sender or "4" if the port is modeled for the receiver, and the name of the receiving component. The same convention is used for interface names containing the name of the sender's type, followed by "2" and the name of the receiver's type.


Idle
cardInserted to User2CardUnit
WaitForAmount
askForAmount
amount(10) to User2Console
WaitForResponse
displayMessage
retrieveCard
cardRemoved to User2CardUnit
WaitForMoney
releaseMoney
Idle
cardInserted
init to CardUnit2Ctr_Init
WaitForEject
ok to CardUnit2Ctr_Eject
ejectCard
retrieveCard to CardUnit2User
/t.set(0)
WaitForRemoval
cardRemoved
[t > 5]
nok to CardUnit2Ctr_Eject
Error
Idle
init
askForAmount to Display2User
amount
displayMessage to Display2User
/t.set(0)
EjectCard
[t = 1]
ejectCard to Ctr2CardUnit
WaitForRemoval
nok
Error
ok
releaseMoney(amount.value) to Dispenser2User
ReleaseMoney

(a) *User*

(b) *CardUnit*

(c) *Controller*

Figure 4.2 – State machines modeling the behavior of the three main blocks of the sATM.

the multi-layer architecture of a system with block definition diagrams and internal block diagrams. Components are deemed to communicate by asynchronous message passing via ports and connectors. Restrictions have been imposed on ports, which

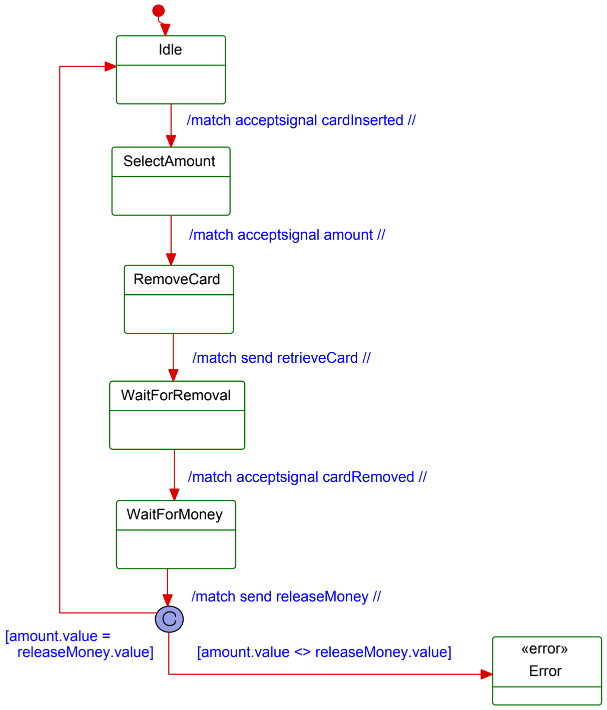

Idle
/match acceptsignal cardInserted //
SelectAmount
/match acceptsignal amount //
RemoveCard
/match send retrieveCard //
WaitForRemoval
/match acceptsignal cardRemoved //
WaitForMoney
/match send releaseMoney //
C
[amount.value = releaseMoney.value]
[amount.value <> releaseMoney.value]
«error»
Error

Figure 4.3 – SysML formalization with an *observer* of the Requirement 4.1: the amount released by the sATM is equal to the amount demanded.

need to be typed with one interface describing the inbound or outbound accepted requests, in order to have a typing system. Moreover, ports can transfer requests to only one target. Finally, the behavior of the system is represented by state machines invoking actions like signal output, variable assignment or expression evaluation.

For modeling real-time systems, we have introduced the OMEGA profile which defines timed behavior as extension via the predefined type Timer. The progress of time can be controlled by two urgency stereotypes — eager and lazy — which allow for a more compact modeling and introduce the formal semantics as timed automata with urgency. The second extension OMEGA describes is the formalization of the requirement with the concept of observer, while in SysML requirements are rather informal. The choice of an automata-based language is more adapted to field engineers than to require a formalization in Linear Temporal Logic or property modeling language. The differences between the observer and the component are minimal, almost the same language being used for both description. Moreover, the formalization of a requirement allows to formally verify its satisfaction by the system design, OMEGA being connected with the IF Toolset.

5 Modeling Behavioral Assume/Guarantee Contracts in SysML

The aim of this work is to use the contract-based reasoning methodology presented in Section 3.1.1 for the compositional design and verification of critical real-time embedded system designs with respect to timed safety properties. We have previously identified the component language as a subset of concepts from SysML required to model such systems. The next step consists in introducing the notion of contract and of the refinement relations they are subject to, into system models by defining a clear and coherent syntax for these concepts.

In Section 5.1, we propose an extension of the component language in the form of a domain meta-model to represent the contract-based meta-theory from [143, 144, 142] that we instantiate. In order to fully comply to the definition of this meta-theory, we provide for each new concept a set of well-formedness rules that will further guarantee the correct application of the contract-based compositional reasoning. A profile instantiation of the meta-model for the OMEGA working context is provided in Section 5.2. Section 5.3 illustrates the application of the entire contract-based methodology on the sATM running example.

5.1 A Meta-Model for Behavioral Contracts[1]

In order to be able to use contracts in a system design, we define in this section a domain meta-model, illustrated in Figure 5.1, that captures the key concepts from the meta-theory of Section 3.1.1. For the sake of the clarity, we define this meta-model on UML; its application to SysML is straightforward by using the same base meta-classes from the *UML4SysML* package [90].

[1] This section is based on [72], [73].

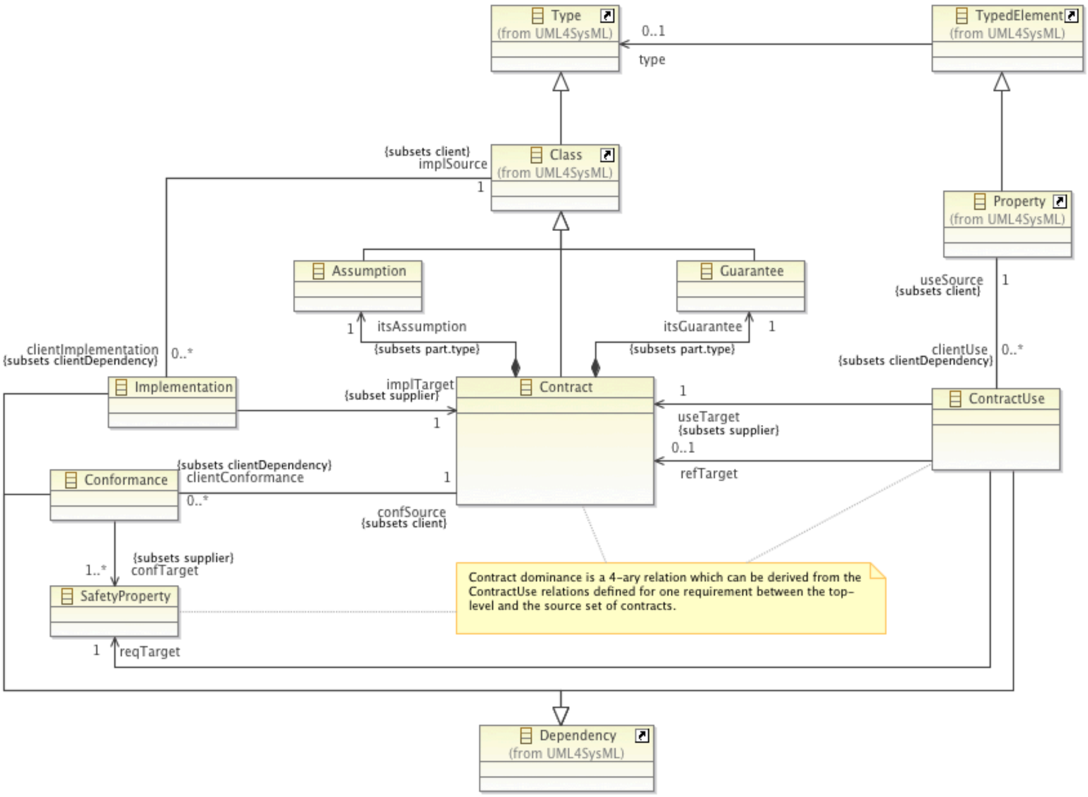


Figure 5.1 – An extension of the UML meta-model for contract-based reasoning.

We start by presenting the meta-classes that are reused as such from the UML standard and which are also members of the *UML4SysML* package. The notion of component (part) which participates in a composite structure is modeled by the meta-class *Property*. The meta-class *Class* denotes the type of components.

In Section 4.1, we have mentioned that we represent the general notion of timed safety requirement by the meta-class *SafetyProperty*. This is due to the fact that we are interested in keeping the notion of formal requirement abstract and defining a generic meta-model that can be applied to other UML/SysML-based component languages, which have the requirements instantiated by other modeling frameworks as described in Chapter 1.

There are two categories of notions that are defined within the meta-theory from Section 3.1.1: (1) those that define how to model a contract represented in the upper part of the meta-model of Figure 5.1 and (2) those that define which verification relations can be used and between which end elements (components and/or contracts) represented in the lower part of Figure 5.1. In the following we describe the notation for each concept and we define a set of well-formedness rules to ensure that the representation complies to the definition of the meta-theory. Moreover, this set of rules has been formalized with OCL [92] which allows to automatically verify (using an OCL interpreter, Topcased[2] in our case) the static typing of a model extended with contracts before applying the verification technique for system behavior. This formalization has the advantage to find from the first system design missing connectors or port typing problems.

We firstly introduce the two concepts that constitute a contract, the assumption and the guarantee, which are respectively represented by the meta-classes *Assumption* and *Guarantee*. Both notions are of type *Class*. The intuition behind this modeling is straightforward: in the meta-theory an assumption/guarantee is a particular kind of component, so it is natural to model it as a class with a behavior modeled by a state machine. Thus, assumptions/guarantees are described by the same language syntax as the system components. However, assumptions/guarantees are presented in the meta-theory as stand-alone components that are not linked via associations or generalizations to other components. Therefore we need to restrict their syntax with respect to the relations they may be involved in: all relations are forbidden except the interface realizations needed to type ports. Yet, the *Assumption*/*Guarantee* may define a composite sub-structure, such an example being provided in the case study of Chapter 9.

Rule 1. *An Assumption has no relations: associations, generalizations, realizations and dependencies are forbidden.*

Rule 2. *A Guarantee has no relations: associations, generalizations, realizations and dependencies are forbidden.*

The OCL formalization for Rule 1 is presented in Listing 5.1 and for Rule 2 in Listing 5.2. The first term of the invariant models that neither the assumption nor guarantee have properties of type class that are not composite (composite structures are allowed) or of type Timer (clocks). The second term verifies that the assumption/guarantee doesn't have any parents, while the third verifies there are no dependencies having them as starting point.

[2] http://www.topcased.org, http://polarsys.org/

Listing 5.1 OCL code for well-formedness of assumption.

```
context Assumption

-- Rule: An assumption has only properties with predefined types (i.e.\ an assumption is not involved in associations, aggregations and compositions)
def: assumptionHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.name<>'Timer' and not a.isComposite)->size() = 0

def: assumptionPropertiesWellFormed : Boolean =
  self.assumptionHasNoPropertiesClassType

-- Rule: An assumption is not involved in any generalization relations (has no parents)
def: assumptionHasNoGenerals : Boolean =
  self.general->size() = 0

-- Rule: An assumption does not depend on any model element
def: assumptionHasNoDependencies : Boolean =
  self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization))->size() = 0

inv assumptionWellFormed : self.assumptionPropertiesWellFormed and self.assumptionHasNoGenerals and self.assumptionHasNoDependencies
```

Listing 5.2 OCL code for well-formedness of guarantee.

```
context Guarantee

-- Rule: A guarantee has only properties with predefined types (i.e.\ an assumption is not involved in associations, aggregations and compositions)
def: guaranteeHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.name<>'Timer' and not a.isComposite)->size() = 0

def: guaranteePropertiesWellFormed : Boolean =
  self.guaranteeHasNoPropertiesClassType

-- Rule: A guarantee is not involved in any generalization relations (has no parents)
def: guaranteeHasNoGenerals : Boolean =
  self.general->size() = 0

```

```
-- Rule: A guarantee does not depend on any model element
def: guaranteeHasNoDependencies : Boolean =
  self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization))->size() = 0

inv guaranteeWellFormed : self.guaranteePropertiesWellFormed and self.guaranteeHasNoGenerals and self.guaranteeHasNoDependencies
```

A contract is represented by the meta-class *Contract* type of *Class* as a composite structure containing exactly one assumption and one guarantee, i.e. all other properties are forbidden. In order to comply to definition of contract from the meta-theory, we need again to restrict the language a contract can use. Therefore, a contract does not exhibit any behavior, i.e. no state machine can be modeled in the contract class, and it is not involved in any other relations than the ones defined in the meta-theory. This modeling of a contract allows for *reusability*: a contract is defined only by instances, while *types* (assumption/guarantee) can be used within other contracts too.

Rule 3. *A Contract does not own any properties (except the composite assumption and guarantee), any operations or signal receptions and any state machines. A Contract is not involved in other relations besides conformance, dominance and contract satisfaction, i.e. associations, generalizations and aggregations/compositions are forbidden.*

Listing 5.3 presents the OCL formalization of Rule 3. The first term of the invariant verifies that the contract does not define any properties except the composite assumption and guarantee. The second and third term model that the contract does not exhibit any behavior, while the fourth eliminates generalization relations for contracts.

Listing 5.3 OCL code for well-formedness of contract structure.

```
context Contract

-- Rule: A contract has no properties besides one part typed assumption and one part typed guarantee (i.e.\ no properties with predefined type and no properties from associations, aggregations or compositions)
def: contractHasNoPropertiesPredefinedType : Boolean =
  self.ownedAttribute->select(a | not a.type.oclIsTypeOf(uml::Class))->size() = 0
def: contractHasNoPropertiesClassType : Boolean =
    self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and
    ((not a.type.oclAsType(uml::Class).isAssumption) or (a.type.oclAsType(uml::Class).isAssumption and not a.isComposite)) and
    ((not a.type.oclAsType(uml::Class).isGuarantee) or (a.type.oclAsType(uml::Class).isGuarantee and not a.isComposite)))->size() = 0
  def: contractPropertiesWellFormed : Boolean =
    self.contractHasNoPropertiesPredefinedType and self.contractHasNoPropertiesClassType

  -- Rule: A contract has no operations
  def: contractHasNoOperations : Boolean =
    self.ownedOperation->size() = 0

  -- Rule: A contract has no statemachine
  def : contractHasNoStateMachine : Boolean =
    self.ownedBehavior->size() = 0

  -- Rule: A contract is not involved in any generalization relation (has no parents)
  def : contractHasNoGenerals : Boolean =
    self.general->size() = 0

  inv contractWellFormed : self.contractPropertiesWellFormed and self.contractHasNoOperations and self.contractHasNoStateMachine and self.contractHasNoGenerals
```

The meta-theory requires for a contract to be a closed component. For the input/output directionality of UML signals it means that all outputs of the contract's assumption/guarantee have their target within the guarantee/assumption and all input triggers are enabled by actions of the other. Since the communication for our system models is based on ports and connectors, we express this constraint on contracts with respect to port types that must be matched with reversed direction.

Rule 4. *Given a Contract, the Assumption and the Guarantee define a closed system: all ports of each type have a correspondent (by type and conjugated direction) within the ports of the other type.*

Moreover, the purpose of a contract is to model a partial behavior with respect to a requirement. We enable this option by allowing a guarantee to define a subset from the signature of the component satisfying the contract. We denote here contract satisfaction by the meta-class *Implementation* and we explain in the following the rationale of this notation. This constraint is also formalized based on port correspondence: a port of the guarantee has to have the same name, type and direction as the port of the class typing the component. We require for the guarantee ports to have the same name as the component ports for compilation reasons. When transformed at the semantical level, we can deduce based on name correspondence and connectors the target for the guarantee's signals. The aim of this rule is to contribute to the requirement-driven design of systems: specifications are refined towards implementations from a requirement, while integrating multiple requirements in the same specification adds, possibly partially disjoint, behaviors that lead to an enlarged component signature.

Rule 5. *Given an Implementation, the set of ports of the contracts Guarantee is included in the set of ports of the component source.*

Listing 5.4 presents the OCL formalization of the previous two rules. For Rule 4 and 5, we have defined two helper functions: *isConjugatedOf* to verify if two ports are conjugated and *isIdenticalTo* to verify if two ports are corresponding. The formalization of Rule 4 consists in verifying that for each port of one type there is at least one port of the other matching the criteria. In order to avoid possible broadcast, we verify that assumption and guarantee have the same number of ports. The formalization of Rule 5 summarizes to iterating the set of ports of the guarantee and verifying that for each port there is one and only one correspondent in the definition of the component.

Listing 5.4 OCL code for well-formedness of contracts.

```
context Port

def: isConjugatedOf(p:Port) : Boolean = self.direction <> p.direction and self.interface = p.interface
def: isIdenticalTo(p:Port) : Boolean = self.name = p.name and self.direction = p.direction and self.interface = p.interface

context Contract

-- Rule: The assumption and guarantee of a contract define a closed system with respect to ports
def: haveIdenticalNoOfPorts : Boolean =
  self.itsAssumption.ownedPort->size() = self.itsGuarantee.ownedPort->size()
def: assumptionPortsSubsetGuaranteePorts : Boolean =
  self.itsAssumption.ownedPort->forAll(p1 | self.itsGuarantee.ownedPort->select(p2| p1.isConjugatedOf(p2))->size() >= 1)
def: guaranteePortsSubsetAssumptionPorts : Boolean =
  self.itsGuarantee.ownedPort->forAll(p1 | self.itsAssumption.
    ownedPort->select(p2 | p1.isConjugatedOf(p2))->size() >= 1)

def: contractAGPortsWellFormed : Boolean =
  self.haveIdenticalNoOfPorts and self.
    assumptionPortsSubsetGuaranteePorts and self.
    guaranteePortsSubsetAssumptionPorts

inv contractClosedSystem : self.contractAGPortsWellFormed

context Implementation

-- Rule: The set of ports of the Guarantee is a subset or equal to
    the set of ports of the Part implementing it
def: guaranteePortsSubsetPartPorts : Boolean =
  self.implTarget.itsGuarantee.ownedPort->forAll(p1 | self.
    implSource.ownedPort->select(p2 | p2.isIdenticalTo(p1))->size()
     = 1)

def: guaranteePortsWellFormed : Boolean =
  self.guaranteePortsSubsetPartPorts

inv implementationGuaranteePortsWellFormed : self.
  guaranteePortsWellFormed
```

As we have mentioned, the satisfaction relation that relates a component to a contract is represented by an *Implementation* at the level of the type of the component. This relation, subtype of *Dependency*, is defined between a *Class* and a *Contract* in Figure 5.1 and expresses that the class satisfies the contract. A contract can be implemented by several classes, while a class can implement several contracts. This summarizes the multi-view modeling aspect of our approach. Then, for verifying one requirement, the designer has to chose in the bouquet of implemented contracts which one guarantees the satisfaction of the requirement. Therefore, we define a second relation also type of *Dependency*, named *ContractUse*, that models contract satisfaction at the components level. Since a system model may have to satisfy several requirements, then for each use of a contract the designer has to specify which requirement is concerned by the current *ContractUse*. This prerequisite is modeled in Figure 5.1 by the unidirectional association from *ContractUse* to *SafetyProperty*.

Since unambiguous typing of models is mandatory, we require that *ContractUse* relations to be modeled between a *Property* and a *Contract* if and only if the property's type implements the contract. We note that given these definitions, the *Implementation* relations can be easily derived from the modeled *ContractUse* relations.

Rule 6. *A Contract can be used by a Property if and only if the property's type implements the contract.*

Listing 5.5 presents the formalization of the rule above. The function *getImplementationsForTarget* computes the set of classes that are modeled as sources in an *Implementation* relation which has as target the target of the *contractUse* relation. Next the function *canContractBeUsed* verifies that the computed set of classes contains the class that types the component using the contract.

Listing 5.5 OCL code for well-formedness of contract usage.

```
context ContractUse

-- Rule: A contract can be used in a proof tree if and only if the type of the property using it implements the contract
def: getImplementationsForTarget : Set(Class) = self.useTarget.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.isImplementation and d.implTarget = useTarget).implSource.oclAsType(uml::Class)->asSet()

def: canContractBeUsed : Boolean = self.getImplementationsForTarget->includes(self.useSource.type.oclAsType(uml::Class))

def: contractUseWellFormed : Boolean = self.canContractBeUsed

inv contractUseWellFormed : self.contractUseWellFormed
```

The second step of the methodology from Section 3.1.1 consists in modeling the *dominance* relation between one, more general, contract and a set of, more specific, contracts. This relation is not explicitly modeled in Figure 5.1 since it can be deduced from the *ContractUse* relations. Indeed, for the meta-theory to be applied, each component of the system — atomic or composed — involved in the satisfaction of a requirement has a *ContractUse* relation to its contract. Then, from the top-level contract and the running requirement we get via *ContractUse* the composed component implementing it. For each of its sub-components, based again on *ContractUse* relations annotated with the requirement under study, we compute the set of source contracts.

However, there is one case in which the definition of *ContractUse* is not sufficient in

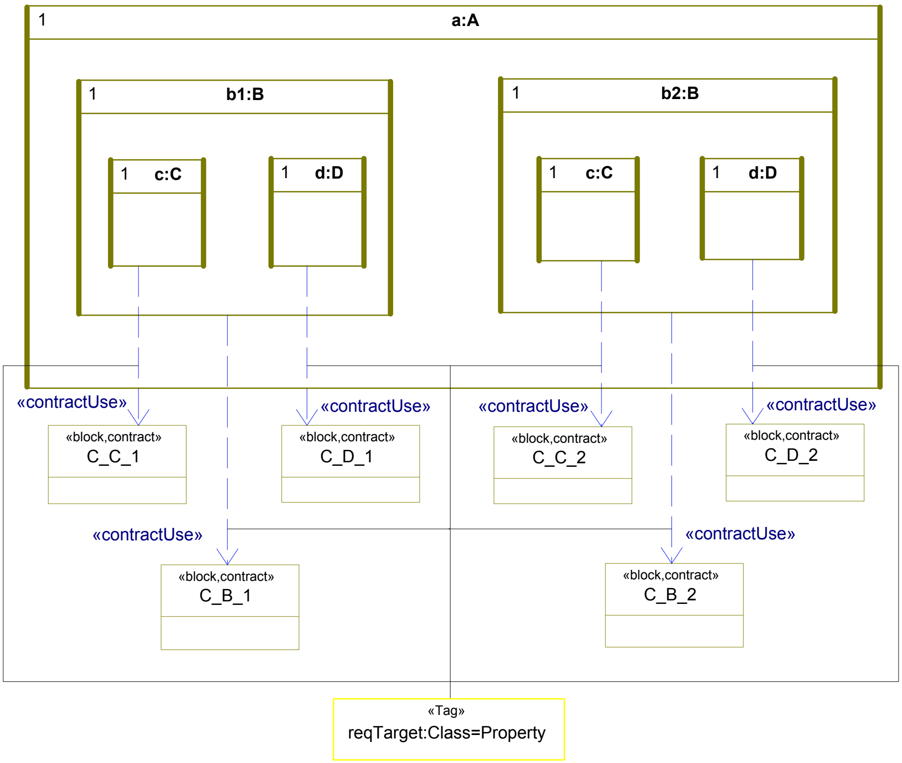


Figure 5.2 – Configuration which shows the necessity for *ContractUse* relation to point to the refined contract.

order to fully determine dominance. An example is presented in Figure 5.2: suppose a component a that contains two components $b1$ and $b2$ of the same type B. At their tour, each sub-component defines a composite structure containing one component c and one component d. For verifying the requirement *Property*, the component $b1$ uses the contract C_B_1 which is dominated by $\{C\_C\_1, C\_D\_1\}$, while the component $b2$ uses the contract C_B_2 dominated by $\{C\_C\_2, C\_D\_2\}$. At the graphical level, dominance seems completely defined by the modeled *ContractUse*. But the dependency relations for sub-components c and d are stored at the level of type B. Then ambiguities are introduced in the (textual) model since we cannot identify in which upper context *ContractUse* relations are defined. As an example, the set $\{C\_C\_2, C\_D\_2\}$ can be computed as dominating contracts for C_B_1, which is incorrect.

Therefore, we need to make the distinction between *contractUse* relations defined in different contexts in order to deduce the correct dominance relation. We require for each *ContractUse* relation to specify in which dominance relation takes part by marking the dominated contract. This condition is represented in the meta-model of Figure 5.1 by the association from *ContractUse* to *Contract* denoted by the role *refTarget*. We remark that this is mandatory only for the particular case in which several instances of the same composite structure are modeled and they use different contracts for the satisfaction of the same requirement. In consequence, dominance is a 4-ary relation which can be fully determined based on *Contract*, *ContractUse* and *SafetyProperty*. Dominance becomes a ternary relation when the same contract is refined once regardless of the number of composite instances.

For the clarity of the presentation, we will assume in the following that all *ContractUse* relations involved in a dominance relation specify the dominated contract whether it is compulsory or not.

Similarly to *Implementation*, the dominance relation is also subject to signature refinement: the signature of the dominated guarantee can be a subset of or equal to the union of signatures of the source guarantees. We express this rule also on ports: informally, each port of the target guarantee must be identical by type and direction to a port from the previously defined source set.

Rule 7. *Given a dominated contract, the set of ports of its guarantee is a subset or equal to the set of ports of the dominating contracts.*

The OCL formalization of this rule is presented in Listing 5.6. We situate Rule 7 in the *Property* context that uses the dominated contract since the source set of contracts has to be computed based on its parts. We consider that a contract is dominated if there is at least one sub-component that has a *ContractUse* relation for the same *SafetyRequirement* and points to the target contract (the *refTarget* value). This condition is described by the *isRefined* function. Then, for each requirement, we compute from the *Property* via its parts (*getPart* function) the set of contracts used in the considered dominance relation (*getUseContractsOfParts* function). We iterate through the source set of contracts guarantees and we build the set of ports defined (*getPortsFromUseContractsOfParts* function). Finally, for each port of the dominated guarantee we verify if there is at least one matching the criteria within the computed set of ports (*refTargetPortsSubsetSoucesPorts* function).

Listing 5.6 OCL code for well-formedness of contract signature refinement in a dominance relation.

```
context Port

def: hasTypeIdenticalTo(p:Port) : Boolean = self.direction = p.
  direction and self.interface = p.interface

```

```
context Class

def: getPart : Set(Property) = self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.isComposite)
def: getUsedContractsOfParts(targetContract:Contract, req:SafetyProperty) : Set(Class) =
    self.getPart->iterate(p:Property; res:Set(Class)=Set{} | res->union(p.clientDependency->select(d | d.isUsage and
        d.reqTarget = req and d.refTarget = targetContract).useTarget).oclAsType(uml::Class))
def: getPortsFromUsedContractsOfParts(targetContract:Contract, req:SafetyProperty) : Set(Port) =
    self.getUsedContractsOfParts(targetContract, req)->iterate(c:Class; res:Set(Port)=Set{} | res->union(c.itsGuarantee.ownedPort))

context Property

def: isUsingContract(targetContract:Contract, req:SafetyProperty) : Boolean = self.clientDependency->select(d | d.isUsage and d.reqTarget = req and d.refTarget = targetContract)->size() > 0
def: getContractUseRelations : Set(Dependency) = self.clientDependency->select(d | d.isUsage)
def: isRefined(targetContract:Contract, req:SafetyProperty) : Boolean =
    self.type.oclIsTypeOf(uml::Class) and self.type.oclAsType(uml::Class).isComposite and
        self.type.oclAsType(uml::Class).ownedAttribute->exists(a | a.type.oclIsTypeOf(uml::Class) and a.isComposite and a.isUsingContract(targetContract, req))

def: refTargetPortsSubsetSourcesPorts(targetContract:Contract, req:SafetyProperty) : Boolean =
    let sp:Set(Port) = self.type.oclAsType(uml::Class).getPortsFromUsedContractsOfParts(targetContract, req) in
        targetContract.itsGuarantee.ownedPort->forAll(p1 | sp->select(p2 | p1.hasTypeIdenticalTo(p2))->size() >= 1)

-- Rule: The set of ports of a dominated guarantee have a correspondent within the set of ports of the dominating guarantees
def: targetGuaranteePortsWellFormed : Boolean =
    self.getContractUseRelations->forAll(d |
        if self.isRefined(d.useTarget, d.reqTarget)
            then self.refTargetPortsSubsetSourcesPorts(d.useTarget, d.reqTarget)
        else
```

```
            true
        endif)

inv refinementTargetGuaranteePortsWellFormed : self.targetGuaranteePortsWellFormed
```

The condition to have at least one port from the source set matching the criteria is due to the fact that the guarantees might be defined for different components having the same type which all model the same port, whereas the dominated guarantee has only one port of that type and direction. Such an example is provided in the case study of Chapter 9.

The last step of the reasoning consists in verifying conformance which is represented in Figure 5.1 by the meta-class *Conformance*. This meta-class, type of *Dependency*, links a *Contract* to a *SafetyProperty*. We note that one contract can serve as source for checking several requirements.

Since our aim is to use the meta-theory of Section 3.1.1 for the verification of system models, we have to guarantee that each modeled step of the reasoning is complete and unique with respect to each requirement. Then, the following three conditions must be satisfied by system models extended with contracts:

1. for a component using several contracts for the satisfaction of the same requirement, there is one and only one *ContractUse* relation between the *Property*, the *SafetyProperty* and the two instances of *Contract*, *useTarget* and *refTarget*;
2. for a component using only a contract for the satisfaction of one requirement, there is one and only one *ContractUse* relation between the *Property*, the *SafetyProperty* and the *Contract*, and
3. within a model, for any *SafetyProperty* there is a contract conforming to it.

The first two rules ensure the uniqueness of a dominance relation in a given context. Indeed, if a component uses a contract for which a correct refinement is provided based on its sub-components there is no need to define a second refinement for the same contract and the same components. This condition enables the reusability of contracts with respect to dominance. Listing 5.7 provides the OCL formalization of these two rules. The *if* branch of the function *isContractUniqueForRequirementandRefinement* verifies that for each component and each requirement there is at most one *ContractUse* relation. If there are several *contractUse* relations (the *else* branch) then we verify that each of them refers to different dominated contracts, i.e. the number of relations must be equal to the number of dominated contracts, duplicates being removed.

Listing 5.7 OCL code for uniqueness of contract-based proof obligations.

```
context Property

def: isUsingContracts : Boolean = self.getContractUseRelations->
  size() > 0

-- Rule: A component can use at most one contract for the
  satisfaction of one requirement and within one dominance
def: isContractUniqueForRequirementAndRefinement : Boolean =
    let r:Set(Dependency) = self.getContractUseRelations in
    if self.type.oclIsTypeOf(uml::Class) and self.isUsingContracts
        then r->forAll(d1 | if r->excluding(d1)->select(d2 | d1.
          reqTarget = d2.reqTarget)->size() = 0 then true else
                r->select(d2 | d1.reqTarget = d2.reqTarget)->size
                  () =
                r->select(d2 | d1.reqTarget = d2.reqTarget).
                  refTarget->asSet()->size() endif)
    else
        true
    endif

inv contractUseUniqueRR : self.
  isContractUniqueForRequirementAndRefinement
```

Listing 5.8 describes the completeness rule: within the set of conformance relations defined in a model there is at least one having as target the current *SafetyProperty*.

Listing 5.8 OCL code for completeness of contract-based proof obligations.

```
context SafetyProperty

-- Rule: All SafetyProperties have a contract conforming to it
def: isVerified : Boolean =
    self.oclAsType(uml::Classifier).getModel().getDependencies->
      select(d | d.isConformance and d.confTarget->includes(self))
      ->size() > 0

inv safetyPropertyIsVerified : self.isVerified
```

This set of rules allows us to generate an unambiguous and sound set of proof obligations whose satisfaction ensures the satisfaction of system's requirements. The entire code which can be applied in an OCL interpreter on system models

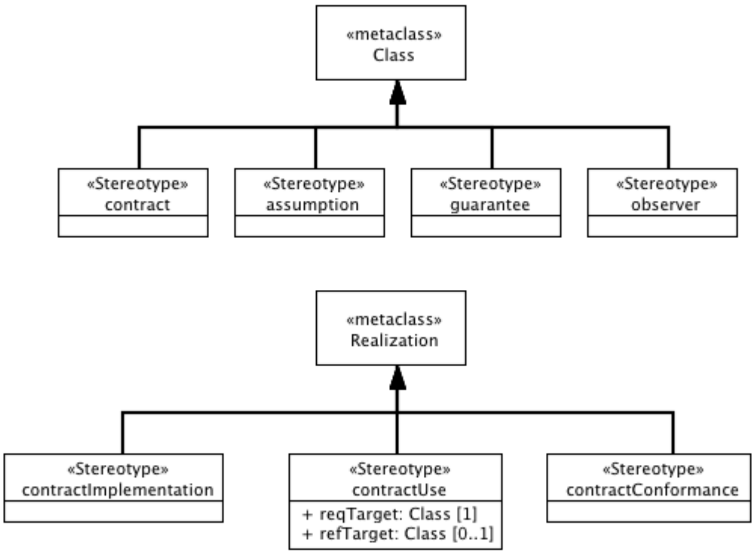


Figure 5.3 – A stereotype implementation of the extended UML meta-model for contract-based reasoning.

extended with contracts can be found in Appendix A.1.

5.2 From Domain Meta-Model to Profile

Using contracts in a standard UML/SysML model implies to instantiate the meta-model previously presented. Our choice is to define a profile and use the stereotype mechanism on the corresponding base meta-classes. For the meta-class *Class* the stereotypes that apply are «*assumption*», «*guarantee*» and «*contract*». Recall that the *SafetyProperty* is modeled in the OMEGA Profile by the «*observer*» stereotype applied on *Class*. For the meta-class *Dependency*, we define the «*contractConformance*», «*contractImplementation*» and «*contractUse*» stereotypes, where the latter has two properties *reqTarget* and *refTarget* (or tagged values depending on the modeler) that refer to the requirement under study, respectively dominated contract, for which the relation is defined. Figure 5.3 resumes the profile-dependent concepts for contract-based reasoning.

We define the type for *reqTarget* and *refTarget* to be *Class* for convenience and we statically verify that the indicated *Class* is correctly stereotyped, «observer» for *reqTarget* and «contract» for *refTarget*. This mechanism allows to consider other formalizations of the requirements in the profile with minor tweaking.

In order to enforce the correct instantiation of the meta-model with stereotypes we

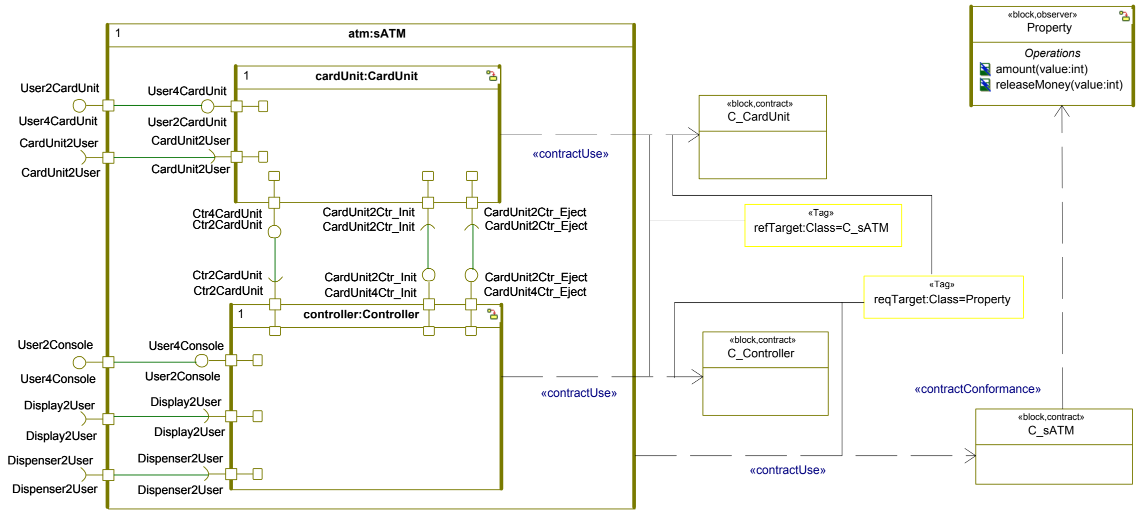


Figure 5.4 – The sATM system model extended with contracts.

propose a second set of well-formedness rules that verify their correct use. This set of rules is detailed in Appendix A.2 and can be statically evaluated on system models.

5.3 Modeling Contracts for the sATM

In the following we describe how we apply the contract-based concepts for the sATM example. The extended system model is presented in Figure 5.4.

Based on the methodology described in Section 3.1.1 and with respect to Requirement 4.1, we identify as system under study S the component *atm*, while the environment E consists in the component *user*. In this case, the system S contains two components: *cardUnit* and *controller*.

From the design perspective, the first step consists in modeling a global contract which has to conform to the requirement *Property*. In consequence, we define a contract class C_sATM and we model a *contractConformance* dependency to *Property*. Since the contract C_sATM is designed to be used in the satisfaction of *Property* by the *atm* component, we model a *contractImplementation* relation between the type *sATM* and the contract and a *contractUse* relation between the component *atm* and its contract for which we specify *Property* as tagged value *reqTarget*. Note that in order to keep Figure 5.4 simple, we do not graphically represent the *contractImplementation* relations.

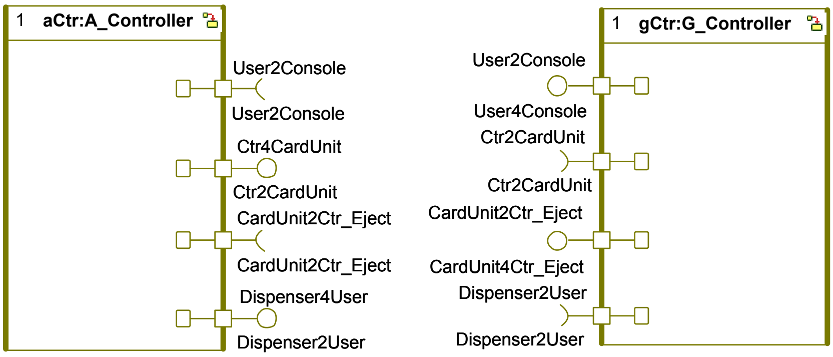


Figure 5.5 – Architecture of the *C_ Controller* contract used by the *controller* component.

The second step consists in modeling two contracts for the *atm*'s sub-components: *C_ Controller* for the *controller* component and *C_ CardUnit* for the *cardUnit* component. We link each component to its corresponding contract via a *contractUse* relation tagged with the same *Property*. Since the set {*C_ Controller*, *C_ CardUnit*} constitutes dominating contracts for *C_ sATM*, we specify the *refTarget* tagged value of the two *contractUse* relations as *C_ sATM*. Then, the modeled contract tree is complete.

Next, we model each contract as a closed composite structure. The behavior of the contained assumptions and guarantees is described in Section 6.2.

The architecture of contract *C_ Controller* is presented in Figure 5.5. Based on the requirement formalization and the behavior of components illustrated in Figure 4.2, we deduce the set of signals, hence the set of ports that need to be modeled, which are directly involved in the satisfaction of the requirement: {*amount*, *ok*, *nok*, *ejectCard*, *releaseMoney*}. Remark that the set of ports of the guarantee is a subset of the set of ports of the *controller*, interaction points with respect to the display function or process initialization being abstracted.

The contract *C_ CardUnit* architecture is modeled in Figure 5.6. Again, we can observe a refinement of signature for the guarantee, which abstracts the initialization process. So, the guarantee signature consists of {*cardInserted*, *ejectCard*, *cardRemoved*, *retrieveCard*, *ok*, *nok*}, contained by the different ports.

Finally, Figure 5.7 presents the architecture of the top-level contract *C_ sATM*. Similarly to the previous contracts we abstract the display function. Then its signature will consist of {*cardInserted*, *amount*, *cardRemoved*, *retrieveCard*, *releaseMoney*}. We observe that the signature of the guarantee is identical to the union

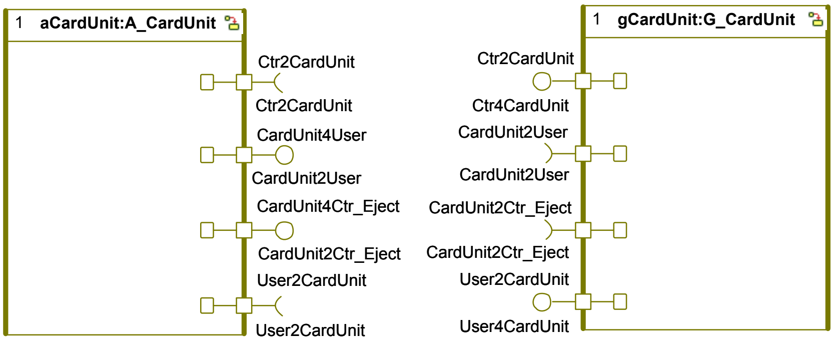


Figure 5.6 – Architecture of the *C_ CardUnit* contract used by the *cardUnit* component.

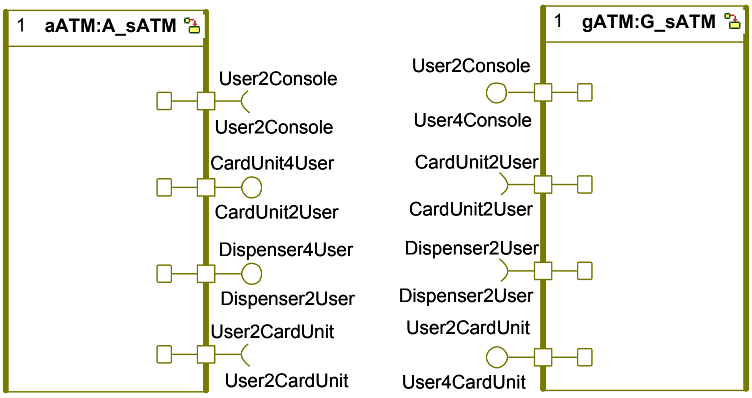


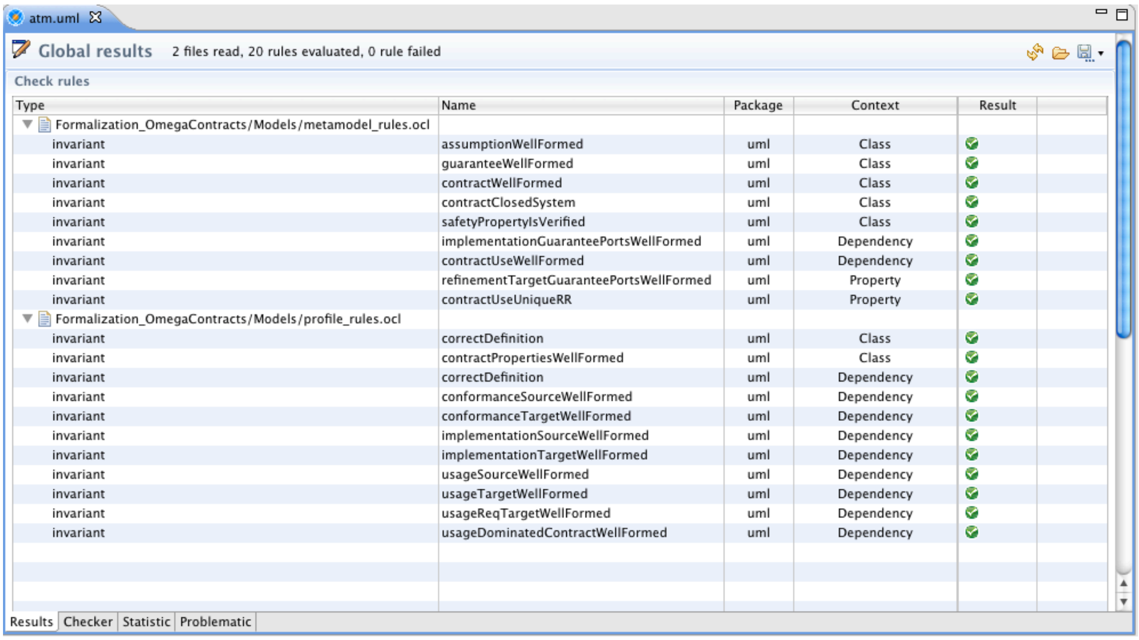
Figure 5.7 – Architecture of the *C_ sATM* contract used by the *atm* component.

of ports of *G_ Controller* and *G_ CardUnit* from which the matched ports (by type and reversed direction) are eliminated.

We verify that the extended system design is correctly modeled. Therefore, we evaluated the sets of well-formedness rules in Topcased and we conclude that the extended sATM model is statically correct with respect to the usage of contract-related notions and the typing system. The result is illustrated in Figure 5.8.

5.4 Conclusion

In this chapter we have defined the contract-related concepts as a domain meta-model which, besides the methodology, takes into consideration fine-grained constraints like contract signature refinement with respect to a component or a set of contracts. These constraints are then expressed and formalized in OCL with the purpose of statically detecting modeling errors, e.g. incompleteness of definitions.



Global results 2 files read, 20 rules evaluated, 0 rule failed

Check rules

Type	Name	Package	Context	Result
▼ Formalization_OmegaContracts/Models/metamodel_rules.ocl				
invariant	assumptionWellFormed	uml	Class	✓
invariant	guaranteeWellFormed	uml	Class	✓
invariant	contractWellFormed	uml	Class	✓
invariant	contractClosedSystem	uml	Class	✓
invariant	safetyPropertyIsVerified	uml	Class	✓
invariant	implementationGuaranteePortsWellFormed	uml	Dependency	✓
invariant	contractUseWellFormed	uml	Dependency	✓
invariant	refinementTargetGuaranteePortsWellFormed	uml	Property	✓
invariant	contractUseUniqueRR	uml	Property	✓
▼ Formalization_OmegaContracts/Models/profile_rules.ocl				
invariant	correctDefinition	uml	Class	✓
invariant	contractPropertiesWellFormed	uml	Class	✓
invariant	correctDefinition	uml	Dependency	✓
invariant	conformanceSourceWellFormed	uml	Dependency	✓
invariant	conformanceTargetWellFormed	uml	Dependency	✓
invariant	implementationSourceWellFormed	uml	Dependency	✓
invariant	implementationTargetWellFormed	uml	Dependency	✓
invariant	usageSourceWellFormed	uml	Dependency	✓
invariant	usageTargetWellFormed	uml	Dependency	✓
invariant	usageReqTargetWellFormed	uml	Dependency	✓
invariant	usageDominatedContractWellFormed	uml	Dependency	✓

Figure 5.8 – Evaluation of the OCL well-formedness set of rules on the sATM system model.

The notion of contract we defined falls under the category of synchronization contracts as described in Section 3.2: the assumption/guarantee has a behavior described by a state machine which corresponds to the call order of requests and their synchronization. We mention that calling our contracts *behavioral* is related to the fact that they describe a behavior and has no connection to the behavioral category of contracts as described in [28]. Still, we consider that our contracts can also be used as syntactical and behavioral ones: the contract signature specifies also the signature of the component and the assumption/guarantee can be considered as a generic view of the pre/post conditions for component use.

To conclude, with respect to the related work presented in Section 3.2 our contract-based theory defined for UML/SysML takes a step further by defining contracts for components, incorporating explicitly modeled behavior as assumption/guarantee in a contract and describing a compositional approach for requirement decomposition and satisfaction which is subject to formal verification.

6 Formal Reasoning with Contracts

Exploiting the contract-based approach for both design and verification of system requirements demands formalizing the semantics of the component language extended with contracts, which includes the refinement relations that contracts are subject to, and proving that the compositional properties required by the meta-theory of [142] hold in our concrete formal framework. In this chapter we define our contract-based theory as an instance of the aforementioned meta-theory, where a component is formalized by a variant of Timed Input/Output Automata from [108] and the contract-related refinement relations are expressed as timed trace inclusion relations.

Section 6.1 describes the formalization of the SysML component language by a variant of Timed Input/Output Automata, since the framework from [108] is not fully adapted to express the semantics of such components. In Section 6.2 we wrap up the contract framework by formalizing the semantics of the refinement relations, denoted proof obligations, and proving that the required compositionality results hold given some restriction on the contract expressiveness. Therefore, we are able to use the additional results the meta-theory offers, like the sufficient condition for proving dominance. Section 6.3 unrolls our instance of reasoning with contracts on the sATM running example. In Section 6.4 we discuss the impact the restrictions from the formal model have on the expressiveness of contracts in SysML. In Section 6.5 we describe and discuss a model-checking approach based on timed property automata and reachability for proving the satisfaction of proof obligations. Finally, we compare our approach with respect to related work in Section 6.6.

6.1 A Flavor of Timed Input/Output Automata for SysML Semantics

The first step of our contribution with respect to the formal framework consists in providing a formalism for the semantics of the SysML component language. Since contracts are defined as a pair of components, the same framework is used to describe the behavior of the assumption/guarantee and how they contribute towards the satisfaction of requirements.

In Section 1.1, we have presented several candidate frameworks for formalizing the semantics of timed reactive components designed with SysML, from which we selected as the most appropriate the Timed Input/Output Automata (TIOA) of [108] due to its thorough definition and the ready-to-use compositional results. However, TIOA as defined in [108] are not fully adapted to express the semantics of SysML components. For example, the input/output matching resulting in an output is not consistent with the unicast communication (i.e. one sender-one receiver) between components, since it provides a pattern of one sender-multiple receivers. Therefore, in the following, we describe a variant of the TIOA framework from [108] in order to correctly encompass the semantics of the component model.

A SysML component is represented at the semantic level by a *timed input/output automaton*:

Definition 6.1 (Timed Input/Output Automaton). A *timed input/output automaton* $\mathcal{A}$ is a tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:

- X is a finite set of discrete variables, Clk a finite set of clocks and $Y = X \cup Clk$ the set of internal variables.
- $Q \subseteq val(Y)$ is a set of states where $val(Y)$ is the set of valuations for Y.
- $\theta \in Q$ is the start state.
- I is a set of input actions, O a set of output actions and V a set of visible actions. We denote by $E = I \cup O \cup V$ the signature of the automaton.
- H is a set of internal actions. $A = E \cup H$ is the set of all executable actions.
- I, O, V and H are pairwise disjoint sets.
- $D \subseteq Q \times A \times Q$ is a set of discrete transitions.
- $\mathcal{T}$ is the set of trajectories. Each trajectory is a function $\tau : J_\tau \to Q$, where J_τ is a real interval of type $[0, d]$ or $[0, \infty)$ with $d \in \mathbb{R}_+$, such that $\forall t \in J_\tau$:
  - $\tau(t)(x) = \tau(0)(x)$, $\forall x \in X$, and
  - $\tau(t)(clk) = \tau(0)(clk) + t$, $\forall clk \in Clk$, i.e. $\dot{\tau}(t)(clk) = 1$.

We note that there are two main differences between Definitions 6.1 and 1.13:

1. The first one relates to the extension of TIOA with *visible* actions, in addition to inputs, outputs and internals. Such actions find their rationale in the output-input matching of components. When computing a composition, in the asynchronous SysML semantics sending and receiving a signal needs to leave a visible trace. In [108], the visibility is achieved by declaring that an input/output match becomes an output, which makes other components susceptible to react to a matched output (e.g. broadcast) and which is not compliant with the component model. Moreover, such visible traces are later exploited for the decomposition and refinement of components towards implementations. The need for visible actions is also motivated by the system requirements which are often described with respect to the visible actions of closed systems.
2. The second difference consists in the restriction of the type of trajectories to constant functions for discrete variables and to linear functions with the derivative equal to 1 for clock variables, while the Definition 1.13 covers general hybrid systems and allows for any functions to be used as trajectories. This restriction makes our timed model expressiveness equivalent to that of Alur-Dill TA [7], thus leaving the possibility of automatically performing reachability analysis or verification of simulation relations. Note that reachability and simulation are undecidable for the TIOA of [108]. Based on this restriction, trajectories are completely defined by their domain which will be solely used in the following to represent the set of trajectories of an automaton. However, the compositionality results required by the meta-theory to hold for sound application are independent from this restriction, i.e. they can be proved also for hybrid systems as described by [108].

Each TIOA offers the following features based on the axioms it satisfies, detailed in Section 1.1.2:

- 0-delay in every state, time continuity and time additivity,
- input-enabledness, i.e. in every state all inputs are available, and
- time elapse enabling, i.e. in every state either time can progress to infinity or to an upper bound in which a locally controlled action $H \cup O \cup V$ is enabled.

We recall here the two notions that allow to describe the behavior of a system: the execution (or path) and the trace. The notation *fval* for a trajectory τ denotes the starting state $\tau(0)$ and *lval* denotes the state for the trajectory's domain supremum.

Definition 6.2 (Execution). An *execution* of a timed input/output automaton $\mathcal{A}$ is a possibly infinite sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$ where

- $\forall i$, $a_i \in A$ and $\tau_i \in \mathcal{T}$,

- $\tau_0.\mathit{fval} = \theta$,
- if τ_i is not the last trajectory in α then τ_i is closed and $\tau_i.\mathit{lval} \xrightarrow{a_{i+1}} \tau_{i+1}.\mathit{fval}$,
- if τ_i is the last trajectory it can be either open or closed, and
- if α is a finite sequence then it ends with a trajectory.

The last item expresses a convenience notation, since an execution can be constituted by only an open trajectory.

An execution fragment can be obtained by replacing the start state θ with a custom given state x. Like trajectories, execution fragments are also closed under countable concatenation.

Definition 6.3 (Trace). Let α be an execution. Then $\mathit{trace}(\alpha)$ is the restriction of α to $(E_{\mathcal{A}}, \emptyset)$, denoted $\mathit{trace}(\alpha) = \alpha\lceil(E_{\mathcal{A}}, \emptyset)$, where:

- each a_i appearing in $\mathit{trace}(\alpha)$ is an action in $E_{\mathcal{A}}$, i.e. removing from α all actions from $H_{\mathcal{A}}$, and
- each $\tau_i : J_{\tau_i} \to \emptyset$, $J_{\tau_i} \subseteq \mathbb{R}_+$, records only the length of time-passage and ignores the evolution of (clock) variables.

Informally, a trace is obtained from an execution by keeping only the length of the time-passage for trajectories, removing actions not in $E_{\mathcal{A}}$ and concatenating all adjacent trajectories. A trace fragment can be obtained from an execution fragment by applying the restriction operator. The formal definition of the restriction operator can be found in [108]. Based on this description we have that the restriction operator is monotone on the set of closed executions fragments, i.e. if α is a prefix of the execution fragment β then $\mathit{trace}(\alpha)$ is a prefix of $\mathit{trace}(\beta)$, and the obtained set is directed with respect to the prefix operator. Therefore, restriction is continuous and the trace of concatenated execution fragments is equal to the concatenation of trace fragments.

We redefine the parallel composition operator for incorporating the extension/restriction we made over the standard Definition 1.13 and allowing the automata to communicate and be executed in parallel. The following definition presents the conditions that have to be satisfied in order to compose two automata.

Definition 6.4 (Compatible components). Two timed input/output automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if for $i, j = \overline{1, 2}$, $i \neq j$, $Y_i \cap Y_j = H_i \cap A_j = V_i \cap A_j = O_i \cap O_j = I_i \cap I_j = \emptyset$.

Besides not sharing the internal variables, internal actions and outputs, which are already required by the standard parallel composition operator, we demand that they have disjoint inputs and visible actions. These conditions ensure that the parallel composition operator corresponds to the SysML semantics.

Syntactically, the parallel composition operator models the input/output synchronization and interleaving of all other unmatched actions.

Definition 6.5 (Parallel composition). If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two compatible timed input/output automata then their *composition* $\mathcal{A}_1 \parallel \mathcal{A}_2$ is defined to be the tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:

- $X = X_1 \cup X_2$ and $Clk = Clk_1 \cup Clk_2$.
- $Q = \{\mathrm{x}_1 \cup \mathrm{x}_2 | \mathrm{x}_1 \in Q_1, \mathrm{x}_2 \in Q_2\}$.
- $\theta = \theta_1 \cup \theta_2$.
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$.
- $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$.
- $V = V_1 \cup V_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$.
- $H = H_1 \cup H_2$.
- D is the set of discrete transitions where for each $\mathrm{x} = \mathrm{x}_1 \cup \mathrm{x}_2$, $\mathrm{x}' = \mathrm{x}'_1 \cup \mathrm{x}'_2 \in Q$ and each $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ if and only if for $i \in \{1, 2\}$, either
  1. $a \in A_i$ and $\mathrm{x}_i \xrightarrow{a}_i \mathrm{x}'_i$, or
  2. $a \notin A_i$ and $\mathrm{x}_i = \mathrm{x}'_i$.
- $\tau \in \mathcal{T} \Leftrightarrow \tau \lceil X_i \in \mathcal{T}_i$, $i \in \{1, 2\}$.

The only difference between this definition and Definition 1.14 is related to the signature of the composite timed input/output automata: the input and output sets of actions consist in those that are not matched between components, while matched inputs/outputs become visible actions. By difference, in Definition 1.14 matched inputs/outputs become outputs, which effectively means that outputs are treated as broadcasts (i.e. one sender-multiple receivers) and is not conform to the usual SysML semantics.

We mention that this definition of parallel composition can encompass asynchronous communications. For this, each automaton owns a predefined internal variable *queue* whose role is to store the received messages. Then, for each input/output match, the receiving automaton adds the message to the queue on transitions with an input. The consumption of the request is modeled by a transition with an internal action ε if it is the head of the queue, i.e. removing the message from the queue. Details about this mechanism are provided in Section 7.1.1.

Theorem 6.1. *$(\mathcal{A}, \parallel)$ is a commutative monoid, where $\mathcal{A}$ denotes the set of TIOA.*

Proof. Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ be three timed input/output automata.

1. *Commutativity*: $\mathcal{A}_1 \parallel \mathcal{A}_2 = \mathcal{A}_2 \parallel \mathcal{A}_1$ is true since the composition operator does not define an order at computation.
2. *Associativity*: By applying the composition operator we obtain $(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3 = \mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3) = (X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:
   - $X = X_1 \cup X_2 \cup X_3$.
   - $Clk = Clk_1 \cup Clk_2 \cup Clk_3$.
   - $Q = \{\mathrm{x}_1 \cup \mathrm{x}_2 \cup \mathrm{x}_3 | \mathrm{x}_1 \in Q_1, \mathrm{x}_2 \in Q_2 \text{ and } \mathrm{x}_3 \in Q_3\}$.
   - $\theta = \theta_1 \cup \theta_2 \cup \theta_3$.
   - $I = (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup (I_3 \setminus (O_1 \cup O_2))$.
   - $O = (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup (O_3 \setminus (I_1 \cup I_2))$.
   - $V = V_1 \cup V_2 \cup V_3 \cup (O_1 \cap (I_2 \cup I_3)) \cup (O_2 \cap (I_1 \cup I_3)) \cup (O_3 \cap (I_1 \cup I_2))$.
   - $H = H_1 \cup H_2 \cup H_3$.
   - D is the set of discrete transitions where for each $\mathrm{x} = \mathrm{x}_1 \cup \mathrm{x}_2 \cup \mathrm{x}_3$, $\mathrm{x}' = \mathrm{x}'_1 \cup \mathrm{x}'_2 \cup \mathrm{x}'_3 \in Q$ and each $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ if and only if for $i \in \{1, 2, 3\}$, either
     (a) $a \in A_i$ and $\mathrm{x}_i \xrightarrow{a} \mathrm{x}'_i$, or
     (b) $a \notin A_i$ and $\mathrm{x}_i = \mathrm{x}'_i$.
   - $\tau \in \mathcal{T} \Leftrightarrow \tau\lceil X_i \in \mathcal{T}_i$, $i \in \{1, 2, 3\}$.
3. The identity element is the *empty* timed input/output automaton: it has no internal variables, it does not perform any actions and can let time elapse to infinity.

The set operator computations for the second item of the proof can be found in Appendix B.1. □

As in [108] we use trace inclusion as the refinement relation between automata, which requires that the two automata have the same signature, i.e. be comparable components.

Definition 6.6 (Comparable components). Two TIOA $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if they have the same signature, formally written $E_1 = E_2$.

Since both the components and the requirements are modeled with the same component language and, in consequence, have their semantics expressed by TIOA, we set the definition of *conformance* from the fourth step of the methodology for verifying that a top contract satisfies a requirement as the trace inclusion relation.

Definition 6.7 (Conformance). Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two comparable TIOA. $\mathcal{A}_1$ *conforms to* (refines) $\mathcal{A}_2$, denoted $\mathcal{A}_1 \preceq \mathcal{A}_2$, if $traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$.

Based on this notation, the fourth step of the reasoning generates the following proof obligation: $\mathcal{C} = (A, G)$ a global contract conforms to φ if $A \parallel G \preceq \varphi$.

Trace inclusion is a preorder relation (i.e. reflexive and transitive) and is preserved by composition, i.e. if $\mathcal{A}_1 \preceq \mathcal{A}_2$ and E is a compatible TIOA with $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{A}_1 \parallel E \preceq \mathcal{A}_2 \parallel E$. The latter property is formalized and proved in the standard framework of TIOA [108] by Theorem 1.1. These results can be easily extended to our variant of TIOA and represent a prerequisite for a sound contract-based framework.

6.2 Contract Theory for TIOA[1]

The second step, described in this section, consists in building the formal contract-based theory on top of the Timed Input/Output Automata (TIOA) framework previously presented by defining the proof obligation generated by each modeled refinement relation and proving the set of compositionality results that ensure the soundness of the method.

A contract is generally represented in Definition 3.1 as a pair of components where the assumption is defined on the environment and the guarantee is defined with respect to the component that uses the contract. We start by clearly stating the notion of contract, where a component is formalized by a TIOA hereafter, as well as some other notions that were not required for the definition of the component framework.

Definition 6.8 (Environment). An *environment Env* for a component K is a timed input/output automaton compatible with K for which the following hold: $I_{Env} \subseteq O_K$ and $O_{Env} \subseteq I_K$.

Definition 6.9 (Closed/Open component). A component K is *closed* if $I_K = O_K = \emptyset$. A component is *open* if it is not closed.

Closed components result from the composition of components having complementary interfaces. Unlike one may expect, we relax here Definition 6.8 to include *partial* environments. This choice is motivated by the layered architecture of systems which needs to sequentially model (refined) components that represent the environment of the other.

Definition 6.10 (Contract). A *contract* $\mathcal{C}$ for a component K is a pair (A, G) of timed input/output automata such that:

[1] This section is based on [76],[74],[75].

- their composition gives a closed system, i.e. $I_A = O_G$ and $I_G = O_A$, and
- the signature of G is a subset of that of K, i.e. $I_G \subseteq I_K$, $O_G \subseteq O_K$ and $V_G \subseteq V_K$.

We denote by the *signature of a contract* the signature of its guarantee.

With respect to the general Definition 3.1 for contracts, Definition 6.10 models compatibility by requiring the composition of the assumption and the guarantee to be closed and the option for a contract to concentrate only on the component's contribution in the satisfaction of the running requirement by a covariant relation on both inputs and outputs, i.e. the component may offer more inputs and outputs than its guarantee. The first item comes naturally from the signature refinement, i.e. a guarantee should know how to handle all requests coming from the environment which may affect the requirement's satisfaction.

We remark here that a correct definition for A over an environment Env implies that $E_A \subseteq E_{Env}$. In consequence the following order relation can be obtained on the signatures of component, environment and contract: $E_G = E_A \subseteq E_{Env} \subseteq E_K$.

Contract satisfaction has been introduced in Definition 3.2 based on a *refinement under context* relation, denoted $\sqsubseteq_{Env}$. In our contract framework, we chose to define refinement under context at its turn based on conformance, in order to respect by default one of its required properties. Since a contract abstracts the signature of a component, the same condition has to be reflected in the definition of refinement under context, i.e. a concrete component K_i has a larger signature than the abstract K_j. Since conformance can be defined only between comparable components, we derive a set of additional timed input/output automata which we compose with the components and the environment such that the comparability condition is satisfied.

Definition 6.11 (Refinement under context). Let K_1 and K_2 be two components such that $I_{K_2} \subseteq I_{K_1} \cup V_{K_1}$, $O_{K_2} \subseteq O_{K_1} \cup V_{K_1}$ and $V_{K_2} \subseteq V_{K_1}$. Let Env be an environment for K_1 compatible with both K_1 and K_2. We say that K_1 *refines K_2 in the context of* Env, denoted $K_1 \sqsubseteq_{Env} K_2$, if

$$K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K' \parallel Env'$$

where K' and Env' are defined such that both members of the conformance relation are closed and comparable, as follows:

- $K' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2})), (V_{K_1} \setminus E_{K_2}), \emptyset, \mathcal{D}_{K'}, 2^{[\mathbb{R}_+^0]})$ where ϕ is the function $\emptyset \to \emptyset$, $\mathcal{D}_{K'} = \{(\phi, a, \phi) | \forall a \in E_{K'}\}$ and $2^{[\mathbb{R}_+^0]} = \{[0, t] | t \in \mathbb{R}_+\} \cup \{[0, \infty)\}$.

- $Env' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \setminus I_{Env}), (I_{K_1} \setminus O_{Env}), \emptyset, \emptyset, \mathcal{D}_{Env'}, 2^{[\mathbb{R}^0_+]})$ where ϕ is the function $\emptyset \to \emptyset$ and $\mathcal{D}_{Env'} = \{(\phi, a, \phi) | \forall a \in E_{Env'}\}$.

Informally, K' is defined as a component that complements the abstract K_2 such that K_1 and $K_2 \parallel K'$ are comparable. In consequence, K' reacts to the actions defined as the set difference between the signatures of K_1 and K_2 and models an abstraction for the part of K_1 that is not involved in the satisfaction of the running requirement. Similarly, Env' is a partial environment for K_1 and $K_2 \parallel K'$ which models actions that do not directly contribute towards the satisfaction of the requirement, in this case represented by K_2. We chose to explicitly represent this additional environment and, therefore, obtain two comparable and closed conformance members. The signature of Env' is given by the actions of the concrete K_1 which are not present in the actions of Env but with reversed directionality. The behavior of K' and Env' is the simplest definition such that all actions are enabled at any moment and time can elapse to infinity, thus modeling a "don't care" semantics.

The particular inclusion relations between the signatures of K_1 and K_2 in the definition are required by the compositionality property of refinement under context, which allows to successively incorporate partial environments in the component under study. Suppose $K_1 = K_1' \parallel K_3$ and $K_2 = K_2' \parallel K_3$, where K_2' is a contract guarantee for K_1', i.e. $I_{K_2'} \subseteq I_{K_1'}$, $O_{K_2'} \subseteq O_{K_1'}$ and $V_{K_2'} \subseteq V_{K_1'}$. Such components are obtained by integrating the partial environment K_3, i.e. $Env = Env' \parallel K_3$. Then, by composition, actions of K_3 may be matched by actions of K_1', but may not have an input/output correspondent in K_2'. This situation also imposes the term $V_{K_1} \cap O_{K_2}$ for inputs of K', since the additional outputs of K_2 may belong to a different component, e.g. K_3, which is not concerned by the satisfaction of the requirement, and the term $V_{K_1} \cap I_{K_2}$ for the outputs of K'.

This definition of refinement under context shows the motivation for adding the set of visible actions for a TIOA. While transforming an input/output match in an output or an input is inconsistent with the intended semantics, the only choice left in the standard definition of TIOA from [108] is to transform it into an internal action. Recall that internal actions are removed from an execution in order to obtain a trace. Therefore, trace inclusion will always hold — a trace reduces to time elapse — and incorrect situations, like performing an unexpected output, cannot be detected.

As summarized in Section 3.1.1, refinement under context has to satisfy several conditions proved in the following in order for the meta-theory to hold.

Theorem 6.2. *Given a set $\mathcal{K}$ of* comparable *components and a fixed environment Env for that signature, the refinement under context $\sqsubseteq_{Env}$ is a preorder over $\mathcal{K}$.*

Proof.

1. *Reflexivity*: $K \sqsubseteq_{Env} K \overset{\Delta}{\Leftrightarrow} K \parallel Env \parallel Env' \preceq K \parallel Env \parallel Env'$ which is true from the definition of the conformance relation.
   K' is not represented since it is the identity element of the composition operator.
2. *Transitivity*: $K_1 \sqsubseteq_{Env} K_2 \wedge K_2 \sqsubseteq_{Env} K_3 \implies K_1 \sqsubseteq_{Env} K_3$.

$$\left.\begin{array}{l} K_1 \sqsubseteq_{Env} K_2 \overset{\Delta}{\Leftrightarrow} K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel Env' \\ K_2 \sqsubseteq_{Env} K_3 \overset{\Delta}{\Leftrightarrow} K_2 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel Env' \end{array}\right\} \overset{Transitivity\ of\ \preceq}{\Longrightarrow}$$

   $\implies K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel Env' \overset{\Delta}{\Leftrightarrow} K_1 \sqsubseteq_{Env} K_3$
   Env' has the same definition since K_1, K_2 and K_3 have the same signature. Similarly, we abstract K_2' and K_3' since they are the identity elements.

□

We remark that the transitivity property holds also if we relax the hypothesis and demand the components to be correctly defined such that refinement under context holds, i.e. the relation is satisfied and the signature definition is refined.

Proposition 6.1. *Let K_1, K_2, K_3 be three components* not necessarily comparable *and Env an environment such that $K_1 \sqsubseteq_{Env} K_2$ and $K_2 \sqsubseteq_{Env} K_3$. Then $K_1 \sqsubseteq_{Env} K_3$.*

Proof. $K_1 \sqsubseteq_{Env} K_2 \overset{\Delta}{\Leftrightarrow} K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K_2' \parallel Env'$ (1)

We write the automaton $Env' = Env_1' \parallel Env_2'$ where :

- $Env_1' = (\emptyset, \emptyset, \{\phi\}, \phi, ((O_{K_1} \cap O_{K_2}) \setminus I_E), ((I_{K_1} \cap I_{K_2}) \setminus O_E), \emptyset, \emptyset, D_{Env_1'}, 2^{[\mathbb{R}^0_+]})$,
- $Env_2' = (\emptyset, \emptyset, \{\phi\}, \phi, ((O_{K_1} \setminus O_{K_2}) \setminus I_E), ((I_{K_1} \setminus I_{K_2}) \setminus O_E), \emptyset, \emptyset, D_{Env_2'}, 2^{[\mathbb{R}^0_+]})$.

Remark that the sets of input and output actions are pairwise disjoint for Env_1' and Env_2'.

We write the automaton $K_2' = K_2'' \parallel Env_3'$ where:

- $K_2'' = (\emptyset, \emptyset, \{\phi\}, \phi, (I_{K_1} \setminus I_{K_2}), (O_{K_1} \setminus O_{K_2}), (V_{K_1} \setminus E_{K_2}), \emptyset, D_{K_2''}, 2^{[\mathbb{R}^0_+]})$,
- $Env_3' = (\emptyset, \emptyset, \{\phi\}, \phi, (V_{K_1} \cap O_{K_2}), (V_{K_1} \cap I_{K_2}), \emptyset, \emptyset, D_{Env_3'}, 2^{[\mathbb{R}^0_+]})$.

Similarly, the sets of inputs, outputs and visible actions are pairwise disjoint for K_2'' and Env_3'.

With this notation:
$(1) \Leftrightarrow K_1 \parallel Env \parallel Env_1' \parallel Env_2' \preceq K_2 \parallel Env \parallel K_2'' \parallel Env_3' \parallel Env_1' \parallel Env_2'$ (2)

$K_2 \sqsubseteq_E K_3 \overset{\Delta}{\Leftrightarrow} K_2 \parallel Env \parallel Env'' \preceq K_3 \parallel Env \parallel K_3' \parallel Env''$ (3)

With the same notation we obtain that $Env'' = Env_1' \parallel Env_3'$, and
$(3) \Leftrightarrow K_2 \parallel E \parallel Env_1' \parallel Env_3' \preceq K_3 \parallel E \parallel K_3' \parallel Env_1' \parallel Env_3'$ (4)

Composing (4) with $K_2'' \parallel Env_2'$ and from Theorem 1.1 we get:
$K_2 \parallel Env \parallel Env_1' \parallel Env_3' \parallel K_2'' \parallel Env_2' \preceq K_3 \parallel Env \parallel K_3' \parallel Env_1' \parallel Env_3' \parallel K_2'' \parallel Env_2' \Leftrightarrow$

$$\left.\begin{aligned} \Leftrightarrow K_2 \parallel Env \parallel K_2'' \parallel Env_3' \parallel Env_1' \parallel Env_2' \preceq K_3 \parallel Env \parallel K_3' \parallel K_2' \parallel \\ \parallel Env_1' \parallel Env_2' \\ (2)\ K_1 \parallel Env \parallel Env_1' \parallel Env_2' \preceq K_2 \parallel Env \parallel K_2'' \parallel Env_3' \parallel \\ \parallel Env_1' \parallel Env_2' \end{aligned}\right\} \xRightarrow{\text{Transitivity of } \preceq}$$

$\Longrightarrow K_1 \parallel Env \parallel Env_1' \parallel Env_2' \preceq K_3 \parallel Env \parallel K_3' \parallel K_2' \parallel Env_1' \parallel Env_2' \Leftrightarrow$
$\Leftrightarrow K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel K_2' \parallel K_3' \parallel Env'$

By denoting $K' = K_2' \parallel K_3'$ we have:
$K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel K' \parallel Env' \overset{\Delta}{\Leftrightarrow} K_1 \sqsubseteq_{Env} K_3$

The last step consists in proving that K' is indeed the automaton generated by the refinement under context relation. Since K_2' and K_3' are built from the hypothesis by the refinement under context relation, by composition they define the correct structure for K'. Moreover:

- $I_{K'} = (I_{K_1} \setminus I_{K_3}) \cup (V_{K_1} \cap O_{K_3})$,
- $O_{K'} = (O_{K_1} \setminus O_{K_3}) \cup (V_{K_1} \cap I_{K_3})$ and
- $V_{K'} = V_{K_1} \setminus E_{K_3}$.

The proofs on the sets of actions for Env'' and K' are detailed in Appendix B.2. □

Furthermore, refinement under context allows deducing conformance for closed systems: it is true from the definition that for any K_1 and K_2 comparable components and Env a complete environment (i.e. $E_{Env} = E_{K_1} = E_{K_2}$) such that $K_1 \sqsubseteq_{Env} K_2$ then $K_1 \parallel Env \preceq K_2 \parallel Env$.

The following theorem that allows for *incremental design* holds in our framework:

Theorem 6.3 (Compositionality). *Let K_1 and K_2 be two components and Env an environment compatible with both K_1 and K_2 such that $Env = Env_1 \parallel Env_2$. Then $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1$.*

Proof. $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel (Env_1 \parallel Env_2) \parallel Env' \preceq K_2 \parallel (Env_1 \parallel Env_2) \parallel K' \parallel Env'$
$K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1 \Leftrightarrow (K_1 \parallel Env_1) \parallel Env_2 \parallel Env'' \preceq (K_2 \parallel Env_1) \parallel Env_2 \parallel K'' \parallel Env''$

The two relations are identical based on the associativity of $\parallel$, where
$Env' = Env'' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \backslash (I_{Env_1} \cup I_{Env_2})), (I_{K_1} \backslash (O_{Env_1} \cup O_{Env_2})), \emptyset, \emptyset, D_{Env'}, 2^{[\mathbb{R}^0_+]})$
and
$K' = K'' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \backslash I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \backslash O_{K_2}) \cup (V_{K_1} \cap I_{K_1})), (V_{K_1} \backslash E_{K_2}), \emptyset, D_{K'}, 2^{[\mathbb{R}^0_+]})$.

The proof with respect to the input, output and visible set of actions is presented in Appendix B.3. □

The soundness of *circular reasoning* is the main result that guarantees the correctness of our contract-based framework and which allows for *independent implementability.*

Theorem 6.4 (Circular reasoning). *Let K be a component, Env its environment and $\mathcal{C} = (A, G)$ a contract for K such that K and G are compatible with both Env and A. If*

1. *$traces_G$ is closed under limits,*
2. *$traces_G$ is closed under time-extension,*
3. *$K \sqsubseteq_A G$ and*
4. *$Env \sqsubseteq_G A$*

then $K \sqsubseteq_{Env} G$.

Proof. Notation: for β be a trace, $\beta\lceil(B, \emptyset)$ denotes the projection of β on the set of actions B.

$K \sqsubseteq_A G \overset{\Delta}{\Leftrightarrow} K \parallel A \parallel A' \preceq G \parallel A \parallel G' \parallel A'$

We write $A' = A'_1 \parallel A'_2$ with

- $A'_1 = (\emptyset, \{\phi\}, \phi, (O_K \setminus I_{Env}), (I_K \setminus O_{Env}), \emptyset, \emptyset, D_{A'_1}, 2^{[\mathbb{R}^0_+]})$,
- $A'_2 = (\emptyset, \{\phi\}, \phi, (I_{Env} \setminus I_A), (O_{Env} \setminus O_A), \emptyset, \emptyset, D_{A'_2}, 2^{[\mathbb{R}^0_+]})$.

This partition of the sets of interfaces is complete due to the relation between the interfaces of the components $A_G = A_A \subseteq A_{Env} \subseteq A_K$. Moreover, the sets of actions are pairwise disjoint.

Similarly, we write $G' = G'_1 \parallel G'_2$ with

- $G'_1 = (\emptyset, \{\phi\}, \phi, (I_K \setminus O_{Env}), (O_K \setminus I_{Env}), (V_K \setminus E_G), \emptyset, D_{G'_1}, 2^{[\mathbb{R}^0_+]})$,
- $G'_2 = (\emptyset, \{\phi\}, \phi, (O_{Env} \setminus I_G), (I_{Env} \setminus O_G), \emptyset, \emptyset, D_{G'_2}, 2^{[\mathbb{R}^0_+]})$

where the sets of actions are again pairwise disjoint.

Then
3. $\overset{\Delta}{\Leftrightarrow} K \parallel A \parallel A'_1 \parallel A'_2 \preceq G \parallel A \parallel G'_1 \parallel G'_2 \parallel A'_1 \parallel A'_2$
4. $\overset{\Delta}{\Leftrightarrow} Env \parallel G \parallel G'_2 \preceq A \parallel G \parallel A'_2 \parallel A'_3 \parallel G'_2$
with $A'_3 = (\emptyset, \{\phi\}, \phi, \emptyset, \emptyset, (V_{Env} \setminus E_A), \emptyset, D_{A'_3}, 2^{[\mathbb{R}^0_+]})$.

With this notation the conclusion becomes: $K \parallel Env \parallel A'_1 \preceq G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$.

We prove this relation in two steps: every closed trace of $K \parallel Env \parallel A'_1$ is a trace of $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$ and every non-closed trace of $K \parallel Env \parallel A'_1$ is a trace of $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$.

Step 1) Every closed trace of $K \parallel Env \parallel A'_1$ is also a trace of $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$. Proof by *induction*.

Step 1.1) Let $\beta \in trajs(\emptyset)$ be a trace of $K \parallel Env \parallel A'_1$. From axiom A0) we have that there is a point trajectory τ_α of G such that $\alpha.ltime = 0$. Since $traces_G$ are closed under time-extension $\implies \alpha^\frown\beta = \beta \in traces_G$ (1)

From the definitions of G'_1 and G'_2 we can similarly deduce that $\beta \in traces_{G'_1}$ (2) and $\beta \in traces_{G'_2}$ (3)

From the hypothesis we have that $\beta \in traces_{K \parallel Env \parallel A'_1} \Leftrightarrow \beta \in traces_{Env \parallel A'_1}$ (4)

(1), (2), (3) and (4) $\implies \beta \in traces_{G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A}$

Step 1.2) Let β' a trace of $K \parallel Env \parallel A'_1$ and $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$. Let

$\beta = \beta'{}^\frown\beta''$ a trace of $K \parallel Env \parallel A'_1$.

We have to prove that $\beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

a) $\beta = \beta' a \tau$ where a is an output action of K and τ a point trajectory.

From the hypothesis we have that $\beta' \in traces_{G\|Env\|G'_1\|G'_2\|A'_1} \implies$

$$\left.\begin{array}{r} \beta' \in traces_{Env\|G\|G'_2} \\ Env \parallel G \parallel G'_2 \preceq A \parallel G \parallel A'_2 \parallel A'_3 \parallel G'_2 \end{array}\right\} \implies \beta' \in traces_{A\|G\|A'_2\|A'_3\|G'_2}$$

$\implies \beta' \lceil (E_A, \emptyset) \in traces_A$ and $\beta' \lceil (E_{A'_2}, \emptyset) \in traces_{A'_2}$.

$I_{Env} \cup I_{A'_1} = O_K$. We have two cases:

i) $a \in I_{A'_1}$. From the hypothesis we have $\beta \in traces_{K\|Env\|A'_1} \implies \beta' \lceil (E_{K\|A'_1}, \emptyset) \in traces_{K\|A'_1}$.
$a \notin I_A \implies \beta \lceil (E_A, \emptyset) = \beta' \lceil (E_A, \emptyset) \in traces_A$.
$a \notin I_{A'_2} \implies \beta \lceil (E_{A'_2}, \emptyset) = \beta' \lceil (E_{A'_2}, \emptyset) \in traces_{A'_2}$

$$\left.\begin{array}{r} \implies \beta \in traces_{K\|A\|A'_1\|A'_2} \\ K \parallel A \parallel A'_1 \parallel A'_2 \preceq G \parallel A \parallel G'_1 \parallel G'_2 \parallel A'_1 \parallel A'_2 \end{array}\right\} \implies$$

$\implies \beta \in traces_{G\|A\|G'_1\|G'_2\|A'_1\|A'_2} \implies \beta \lceil (E_{G\|G'_1\|G'_2}, \emptyset) \in traces_{G\|G'_1\|G'_2}$ (5)
$\beta \in traces_{K\|Env\|A'_1} \implies \beta \lceil (E_{Env\|A'_1}, \emptyset) \in traces_{Env\|A'_1}$ (6)
(5) and (6) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

ii) $a \in I_{Env}$, $I_{Env} = I_A \cup I_{A'_2}$. We have two cases:

ii.1) $a \in I_A$. Let α be an execution of A such that $trace(\alpha) = \beta' \lceil (E_A, \emptyset)$. From the axiom A4) we have that $\exists \mathrm{x}'$ state such that $(\alpha(\alpha.ltime), a, \mathrm{x}')$ is a discrete transition $\implies$
$\implies \beta' \lceil (E_A, \emptyset)^\frown a \lceil (E_A, \emptyset)^\frown \tau \lceil (E_A, \emptyset) = \beta \lceil (E_A, \emptyset) \in traces_A$.
$a \notin I_{A'_2} \implies \beta \lceil (E_{A'_2}, \emptyset) = \beta' \lceil (E_{A'_2}, \emptyset) \in traces_{A'_2}$
From the hypothesis we have $\beta \in traces_{K\|Env\|A'_1} \implies \beta \lceil (E_{K\|A'_1}, \emptyset) \in traces_{K\|A'_1}$

$$\left.\begin{array}{r} \implies \beta \in traces_{K\|A\|A'_1\|A'_2} \\ K \parallel A \parallel A'_1 \parallel A'_2 \preceq G \parallel A \parallel G'_1 \parallel G'_2 \parallel A'_1 \parallel A'_2 \end{array}\right\} \implies$$

$\implies \beta \in traces_{G\|A\|G'_1\|G'_2\|A'_1\|A'_2} \implies \beta \lceil (E_{G\|G'_1\|G'_2}, \emptyset) \in traces_{G\|G'_1\|G'_2}$ (7)
(6) and (7) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

ii.2) $a \in I_{A'_2}$. Let α be an execution of A'_2 such that $trace(\alpha) = \beta' \lceil (E_{A'_2}, \emptyset)$.

From the axiom A4) we have that $\exists \mathrm{x}'$ state such that $(\alpha(\alpha.ltime), a, \mathrm{x}')$ is a discrete transition $\implies \beta'\lceil(E_{A_2'}, \emptyset)^\frown a\lceil(E_{A_2'}, \emptyset)^\frown \tau\lceil(E_{A_2'}, \emptyset) = \beta\lceil(E_{A_2'}, \emptyset) \in$ $traces_{A_2'}$.
$a \notin I_A \implies \beta\lceil(E_A, \emptyset) = \beta'\lceil(E_A, \emptyset) \in traces_A$
From the hypothesis we have $\beta \in traces_{K\|Env\|A_1'} \implies \beta\lceil(E_{K\|A_1'}, \emptyset) \in$ $traces_{K\|A_1'}$

$$\left.\begin{array}{r}\implies \beta \in traces_{K\|A\|A_1'\|A_2'} \\ K \parallel A \parallel A_1' \parallel A_2' \preceq G \parallel A \parallel G_1' \parallel G_2' \parallel A_1' \parallel A_2'\end{array}\right\} \implies$$

$\implies \beta \in traces_{G\|A\|G_1'\|G_2'\|A_1'\|A_2'} \implies \beta\lceil(E_{G\|G_1'\|G_2'}, \emptyset) \in traces_{G\|G_1'\|G_2'}$ (8)
(6) and (8) $\implies \beta \in traces_{G\|Env\|G_1'\|G_2'\|A_1'}$

b) $\beta = \beta' a \tau$ where a is an output action of K and τ a point trajectory.
$\beta' \in traces_{G\|Env\|G_1'\|G_2'\|A_1'} \implies \beta'\lceil(E_G, \emptyset) \in traces_G$, $\beta'\lceil(E_{G_1'}, \emptyset) \in traces_{G_1'}$ and $\beta'\lceil(E_{G_2'}, \emptyset) \in traces_{G_2'}$

$I_G \cup I_{G_2'} = O_{Env}$. We have two cases:

i) $a \in I_G$. Let α be an execution of G such that $trace(\alpha) = \beta'\lceil(E_G, \emptyset)$.
From axiom A4) we have that $\exists \mathrm{x}'$ state with $(\alpha(\alpha.ltime), a, \mathrm{x}')$ discrete transition $\implies \beta'\lceil(E_G, \emptyset)^\frown a\lceil(E_G, \emptyset)^\frown \tau\lceil(E_G, \emptyset) = \beta\lceil(E_G, \emptyset) \in traces_G$ (9)
$a \notin I_{G_2'} \implies \beta\lceil(E_{G_2'}, \emptyset) = \beta'\lceil(E_{G_2'}, \emptyset) \in traces_{G_2'}$ (10)
$a \notin I_{G_1'} \implies \beta\lceil(E_{G_1'}, \emptyset) = \beta'\lceil(E_{G_1'}, \emptyset) \in traces_{G_1'}$ (11)
(6), (9), (10) and (11) $\implies \beta \in traces_{G\|Env\|G_1'\|G_2'\|A_1'}$

ii) $a \in I_{G_2'}$. Let α be an execution of G such that $trace(\alpha) = \beta'\lceil(E_{G_2'}, \emptyset)$.
From axiom A4) we have that $\exists \mathrm{x}'$ state with $(\alpha(\alpha.ltime), a, \mathrm{x}')$ discrete transition $\implies \beta'\lceil(E_{G_2'}, \emptyset)^\frown a\lceil(E_{G_2'}, \emptyset)^\frown \tau\lceil(E_{G_2'}, \emptyset) = \beta\lceil(E_{G_2'}, \emptyset) \in traces_{G_2'}$ (12)
$a \notin I_G \implies \beta\lceil(E_G, \emptyset) = \beta'\lceil(E_G, \emptyset) \in traces_G$ (13)
$a \notin I_{G_1'} \implies \beta\lceil(E_{G_1'}, \emptyset) = \beta'\lceil(E_{G_1'}, \emptyset) \in traces_{G_1'}$ (14)
(6), (12), (13) and (14) $\implies \beta \in traces_{G\|Env\|G_1'\|G_2'\|A_1'}$

c) $\beta = \beta' a \tau$ where a is an output action of A_1' and τ is a point trajectory.
$\beta' \in traces_{G\|Env\|G_1'\|G_2'\|A_1'} \implies \beta'\lceil(E_G, \emptyset) \in traces_G$, $\beta'\lceil(E_{G_1'}, \emptyset) \in traces_{G_1'}$ and $\beta'\lceil(E_{G_2'}, \emptyset) \in traces_{G_2'}$
$a \notin I_G \implies \beta\lceil(E_G, \emptyset) = \beta'\lceil(E_G, \emptyset) \in traces_G$ (15)
$a \in I_{G_1'} (= O_{A_1'})$. Let α be an execution of G_1' such that $trace(\alpha) = \beta'\lceil(E_{G_1'}, \emptyset)$. From the axiom A4) we have that $\exists \mathrm{x}'$ state such that $(\alpha(\alpha.ltime), a, \mathrm{x}')$ is a discrete transition $\implies$
$\implies \beta'\lceil(E_{G_1'}, \emptyset)^\frown a\lceil(E_{G_1'}, \emptyset)^\frown \tau\lceil(E_{G_1'}, \emptyset) = \beta\lceil(E_{G_1'}, \emptyset) \in traces_{G_1'}$ (16)

$a \notin I_{G'_2} \implies \beta\lceil(E_{G'_2}, \emptyset) = \beta'\lceil(E_{G'_2}, \emptyset) \in traces_{G'_2}$ (17)

(6), (15), (16) and (17) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

d) $\beta = \beta' a \tau$ where a is a visible action of K and τ is a point trajectory.
$\beta \in traces_{K\|Env\|A'_1} \implies \beta\lceil(E_K, \emptyset) \in traces_K$
$a \in V_K \implies a \notin E_A$, $a \notin E_{A'_1}$ and $a \notin E_{A'_2}$ and from the hypothesis $\beta'\lceil(E_{A\|A'_1\|A'_2}, \emptyset) \in traces_{A\|A'_1\|A'_2} \implies \beta\lceil(E_{A\|A'_1\|A'_2}, \emptyset) \in traces_{A\|A'_1\|A'_2}$

$$\left.\begin{array}{r} \implies \beta \in traces_{K\|A\|A'_1\|A'_2} \\ K \parallel A \parallel A'_1 \parallel A'_2 \preceq G \parallel A \parallel G'_1 \parallel G'_2 \parallel A'_1 \parallel A'_2 \end{array}\right\} \implies$$

$\implies \beta \in traces_{G\|A\|G'_1\|G'_2\|A'_1\|A'_2} \implies \beta\lceil(E_{G\|G'_1\|G'_2}, \emptyset) \in traces_{G\|G'_1\|G'_2}$ (18)

(6) and (18) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

e) $\beta = \beta' a \tau$ where a is a visible action of Env and τ is a point trajectory.
$\beta' \in traces_{G\|Env\|G'_1\|G'_2} \implies \beta'\lceil(E_{G\|G'_1\|G'_2}, \emptyset) \in traces_{G\|G'_1\|G'_2}$
Since $a \in V_{Env} \implies a \notin E_G$, $a \notin E_{G'_1}$ and $a \notin E_{G'_2} \implies \beta\lceil(E_{G\|G'_1\|G'_2}, \emptyset) = \beta'\lceil(E_{G\|G'_1\|G'_2}, \emptyset) \in traces_{G\|G'_1\|G'_2}$ (19)

(6) and (19) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$

Step 2) Every non-closed trace of $K \parallel Env \parallel A'_1$ is also a trace of $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$. Let β be a non-closed trace of $K \parallel Env \parallel A'_1$. Then β is the limit of a sequence $\beta_1\beta_2\cdots$ of closed traces of $K \parallel Env \parallel A'_1$. We have shown that β_i is closed trace of $K \parallel Env \parallel A'_1$, for all i, and thus β_i is also a trace of $G \parallel Env \parallel G'_1 \parallel G'_2 \parallel A'_1$.

$$\left.\begin{array}{r} \beta_i \in traces_{G\|Env\|G'_1\|G'_2\|A'_1} \implies \beta_i\lceil(E_G, \emptyset) \in traces_G, \forall i \\ \text{restriction is a continuous operation} \end{array}\right\} \implies$$

$$\left.\begin{array}{r} \implies \beta\lceil(E_G, \emptyset) = lim \beta_i\lceil(E_G, \emptyset) \\ traces_G \text{ are closed under limits} \end{array}\right\} \implies \beta\lceil(E_G, \emptyset) \in traces_G (20)$$

Similarly, $\beta\lceil(E_{G'_1}, \emptyset) \in traces_{G'_1}$ (21) and $\beta\lceil(E_{G'_2}, \emptyset) \in traces_{G'_2}$ (22)

(6), (21), (22) and (23) $\implies \beta \in traces_{G\|Env\|G'_1\|G'_2\|A'_1}$ □

This theorem is similar to Theorem 1.2, while the same reasoning is used for the two theorem proofs. While we prove that $K \sqsubseteq_{Env} G$, Theorem 1.2 states that $K \parallel Env \preceq G \parallel A$. Even if the latter is a stronger result with respect to the state space reduction at verification, it doesn't guarantee the circular reasoning needed for contract refinement. Moreover, our conclusion also allows to relax the theorem's hypothesis by requiring only G to model a safety property. Recall that the two conditions on the set of traces — closure under limits and under time-extension — specify that the component is a safety property.

The second step of the contract-based reasoning consists in proving the refinement relation between contracts in order to discard components from now on. Since our theory satisfies the compositionality results required by the meta-theory, we can use the following *sufficient condition for dominance* which is a variant of Theorem 3.1 proved on the general notation of the meta-theory in [142]. Then verifying dominance consists in checking several satisfaction relations on abstract timed input/output automata that are easier to handle.

Theorem 6.5. *$\{C_i\}_{i=1}^n$ dominates C if, for all i, $traces_{G_i}$ and $traces_G$ are closed under limits and under time-extension and*

$$\begin{cases} G_1 \parallel ... \parallel G_n \sqsubseteq_A G \\ A \parallel G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n \sqsubseteq_{G_i} A_i, \ \forall i \end{cases}$$

Proof. Let K_i, $i = \overline{1,n}$, a set of components such that:
(1) $K_i \sqsubseteq_{A_i} G_i$
(2) $G_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$
(3) $A \parallel G_1 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i$

We have to prove that $K_1 \parallel K_2 \parallel \ldots \parallel K_n \sqsubseteq_A G$.

The proof is realized by induction on j where $j = \overline{0,n}$ is the number of guarantees replaced by their corresponding component.

More precisely, we will prove by induction that $K_1 \parallel ... \parallel K_{j-1} \parallel G_j \parallel ... \parallel G_n \sqsubseteq_A G$.

In parallel, we will also need to prove that $A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i$, $\forall i > j$.

Step $j = 0$. Then the conclusion becomes

$G_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$ which is true by hypothesis (2).

Step $j = 1$.

$$\left.\begin{array}{r} \text{From (1) for i=1} \implies K_1 \sqsubseteq_{A_1} G_1 \\ \text{From (3) for i=1} \implies A \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_{G_1} A_1 \end{array}\right\} \overset{Theorem\ 6.4}{\Longrightarrow}$$

$\implies K_1 \sqsubseteq_{A \parallel G_2 \parallel \ldots \parallel G_n} G_1$ (4)

$$\left.\begin{array}{r} (4) \overset{Theorem\ 6.3}{\Longrightarrow} K_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G_1 \parallel G_2 \parallel \ldots \parallel G_n \\ (2)\ G_1 \parallel \ldots \parallel G_n \sqsubseteq_A G \end{array}\right\} \overset{Transitivity\ of\ \sqsubseteq_A}{\Longrightarrow}$$

$\implies K_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$ (5)

$$\left.\begin{array}{r} (4) \overset{Theorem\ 6.3}{\Longrightarrow} A \parallel K_1 \parallel G_2 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel \\ \parallel G_n \sqsubseteq_{G_i} A \parallel G_1 \parallel G_2 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n, \forall i > 1 \\ (3)\ A \parallel G_1 \parallel \ldots G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i \end{array}\right\} \overset{Transitivity\ of\ \sqsubseteq_{G_i}}{\Longrightarrow}$$

$\implies A \parallel K_1 \parallel G_2 \parallel \ldots G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i$, $\forall i > 1$ (6)

Relations (5) and (6) constitute the hypotheses for the induction step at $j = 2$.

Induction step. Let j be fixed. The induction hypotheses for this step are:
$K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_n \sqsubseteq_A G$ (7)
$A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i$, $\forall i > j$ (8)

Then we want to prove that:
$K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel G_n \sqsubseteq_A G$ (9) and
$A \parallel K_1 \parallel \ldots \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j+1$ (10)

We proceed as follows:

$$\left.\begin{array}{r} (1)\ K_{j+1} \sqsubseteq_{A_{j+1}} G_{j+1} \\ \text{From (8) for i=j+1} \implies A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel \\ \parallel G_{j+2} \parallel \ldots \parallel G_n \sqsubseteq_{G_{j+1}} A_{j+1} \end{array}\right\} \overset{Theorem\ 6.4}{\Longrightarrow}$$

$\Longrightarrow K_{j+1} \sqsubseteq_{A\|K_1\|\ldots\|K_j\|G_{j+2}\|\ldots\|G_n} G_{j+1}$ (11)

$$\left.\begin{array}{r} (11) \overset{\mathit{Theorem}\ 6.3}{\Longrightarrow} K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_n \sqsubseteq_A K_1 \parallel \\ \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_n \\ (7)\ K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_n \sqsubseteq_A G \end{array}\right\} \overset{\mathit{Transitivity\ of}\ \sqsubseteq_A}{\Longrightarrow}$$

$\Longrightarrow K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel G_n \sqsubseteq_A G$ (9)

$$\left.\begin{array}{r} (11) \overset{\mathit{Theorem}\ 6.3}{\Longrightarrow} A \parallel K_1 \parallel \ldots \parallel\parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel \\ \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A \parallel K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel \\ \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n, \forall i > j+1 \\ (8)\ A \parallel K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel \\ \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j+1 \end{array}\right\} \overset{\mathit{Transitivity\ of}\ \sqsubseteq_{G_i}}{\Longrightarrow}$$

$\Longrightarrow A \parallel K_1 \parallel \ldots \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j+1$ (10)

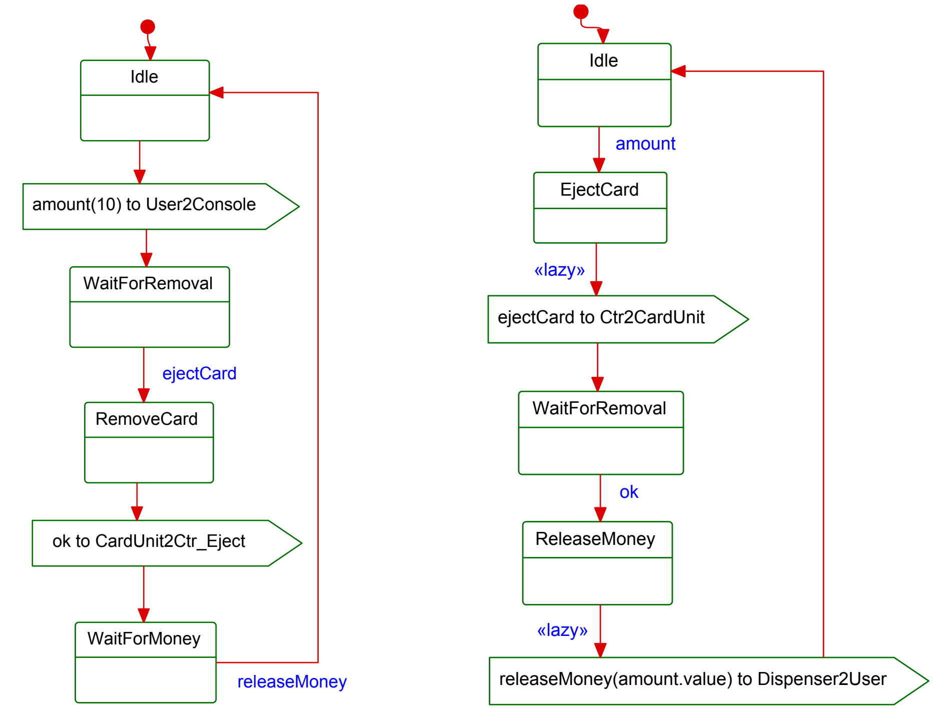
Step $j = n$. From (9), for $j = n$, we obtain $K_1 \parallel K_2 \parallel \ldots \parallel K_n \sqsubseteq_A G$ which implies dominance. □

The previous proof is similar to that of Theorem 3.1 presented in [142] by adapting it to our notation and taking into consideration the compositional results of Theorems 6.2, 6.3, 6.4 and Proposition 6.1.

6.3 Application of the Contract Framework on the sATM

In the sequel we show how our contract framework can be applied on the sATM running example described in Section 4.2 for the satisfaction of Requirement 4.1. We start by describing the set of contracts that will be used for verification; we mention that they were obtained after applying a CEGAR method [50, 29, 1, 97] on proof obligations as explained below.

Figure 6.1 presents the contract's *C_ Controller* assumption and guarantee behavior, modeled for the *controller* component. We have previously identified based on the requirement formalization and the component behaviors the signature of the contract to be {*?amount*, *?ok*, *?nok*, *!ejectCard*, *!releaseMoney*}, where the

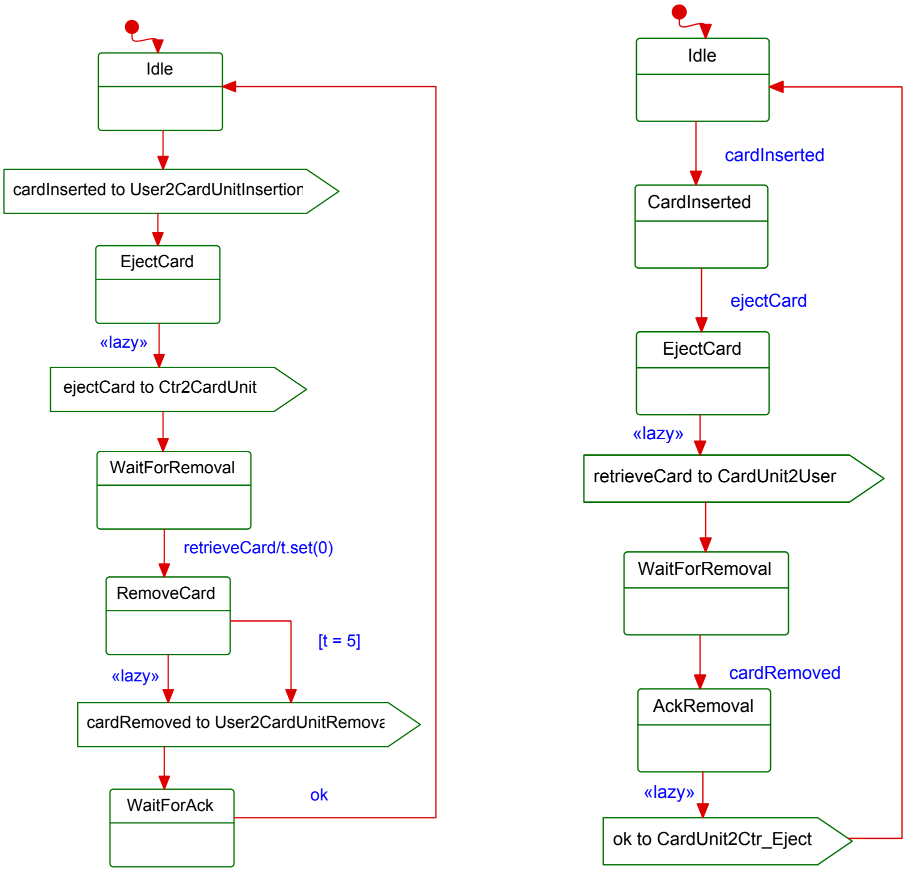


(a) Behavior of the assumption (b) Behavior of the guarantee

Figure 6.1 – Contract modeling for the *controller* component.

initialization process and display role are abstracted. Therefore, the assumption represented in Figure 6.1(a) models that after the *amount* is selected, the card is removed without the occurrence of an error (i.e. the *ok* sendAction). The component guarantees that if the amount is eventually released then it will have the same value as the one demanded by the customer, modeled in Figure 6.1(b) by the parameter *amount.value* of *releaseMoney*.

The contract for the *cardUnit* component is defined over the signature $\{?cardInserted$, $?ejectCard$, $?cardRemoved$, $!retrieveCard$, $!ok$, $!nok\}$. Even though the initialization of the withdrawal process is not concerned by the Requirement 4.1, the card insertion action has to be modeled. We explain the rationale for this request on the contract satisfaction proof. The assumption $A\_CardUnit$ modeled in Figure 6.2(a) is informally described by the requirement: once the signal *retrieveCard* is handled, a *Timer t* is set to 0; then the card is removed within at most 5 time units. The delay is represented with the transition without a guard from *RemoveCard* to the sendAction which has a lazy semantics, while the time guarded transition enforces the execution of the action at 5 time units if not performed before. The *cardUnit* will guarantee in Figure 6.2(b) that only the *ok* signal is raised during the remove process thus eliminating the *nok* branch.



(a) Behavior of the assumption (b) Behavior of the guarantee

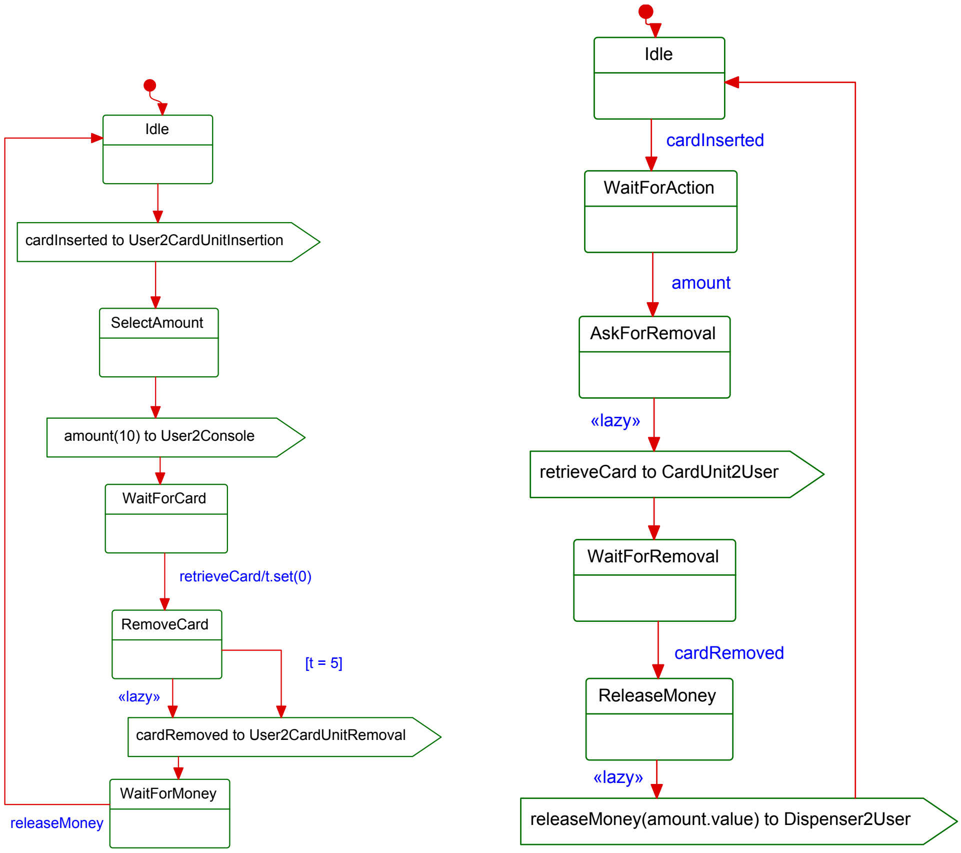
Figure 6.2 – Contract modeling for the *cardUnit* component.

Finally, the top-level contract C_ATM has its signature consisting in $\{?cardInserted$, $?amount$, $?cardRemoved$, $!retrieveCard$, $!releaseMoney\}$ which is identical on inputs and outputs with the signature of $G\_Controller$ and $G\_CardUnit$ composition. The assumption, modeled in Figure 6.3(a), describes a behavior similar to the $A\_CardUnit$. The guarantee, represented in Figure 6.3(b), expresses that if the amount is released then it will have the same value as the one demanded by the customer.

The first step of the reasoning consists in verifying the following contract satisfaction relations:

(1.1) $controller \sqsubseteq_{aCtr} gCtr$,
(1.2) $cardUnit \sqsubseteq_{aCardUnit} gCardUnit$.

The conformance proof obligation generated by the contract satisfaction of $C\_CardUnit$ is written:



(a) Behavior of the assumption

(b) Behavior of the guarantee

Figure 6.3 – Contract modeling for the *atm* component.

$$cardUnit \parallel aCardUnit \parallel addEnvCardUnit \preceq$$
$$gCardUnit \parallel aCardUnit \parallel addGuaCardUnit \parallel addEnvCardUnit$$

where the signature of the additional components *addEnvCardUnit* and *addGuaCardUnit* is the signal *init*. Suppose now that the *cardInserted* signal is not modeled by *aCardUnit* and *gCardUnit*. Then, this signal will appear in the component *addEnvCardUnit* which can execute it at any moment. For example, this message may be sent to the *CardUnit* while it is in the state *WaitForRemoval* and before the occurrence of *cardRemoved* — see Figure 4.2(b). Then *cardUnit* will emit a signal *nok* — its queue is blocked and the second request cannot be handled —, which violates the guarantee. In consequence, *cardInserted* needed to be considered in the modeling of the assumption/guarantee. This reasoning is consistent with the CEGAR method [50, 29, 1, 97] for spurious counterexamples, which allows to perform abstraction refinement.

For the dominance step, we apply Theorem 6.5 since *G_ ATM*, *G_ Controller* and *G_ CardUnit* satisfy the closure under limits and under time-extension conditions and we obtain the following contract satisfaction proofs:

(2.1) $gCtr \parallel gCardUnit \sqsubseteq_{aATM} gATM$,
(2.2) $aATM \parallel gCardUnit \sqsubseteq_{gCtr} aCtr$ and
(2.3) $aATM \parallel gCtr \sqsubseteq_{gCardUnit} aCardUnit$.

Two remarks need to be made here, with respect to the modeling of *cardInserted* in the contract of the *satm* and the lazy stereotypes on assumption transitions. Regarding the *cardInserted* requests, its modeling is imposed by the proof obligation (2.3): in order for refinement under context to be defined the actions of *aCardUnit* need to be a subset of the actions of $aATM \parallel gCtr$. In consequence, the output of *cardInserted* has to be added to the modeling of *aATM* and *gATM*.

With respect to the lazy semantics of some transitions within the assumptions (e.g. *!ejectCard*, *!ok*), this is due to the fact that they also appear in the guarantee of the other component where they have a lazy semantics and individual assumption have to be refined within dominance by the guarantees and the global assumption. For example, in the proof obligation (2.2), the *ok* action of *gCardUnit* is lazy, which imposes the same timed semantics for its correspondence in *aCtr*. This case can also be detected using a counterexample diagnosis based on time elapse.

The third step consists in the satisfaction of the "mirror" *C_ ATM* contract. The next proof obligation has to be satisfied:

(3) $user \sqsubseteq_{gATM} aATM$

We remark that *aATM* is a loose abstraction of the behavior of the *user* where the *cardRemoved* signal has a delay of 5 time units instead of being eager as modeled in the *user* state machine.

The last proof obligation corresponds to the conformance step:

(4) $aATM \parallel gATM \preceq Property.$

6.4 Contract Expressiveness for SysML Models

This theory can be applied on system models extended with contracts if the component playing the role of the guarantee satisfies two important conditions: *closure under limits* and *under time-extension*. We discuss here the restrictions that are imposed by these constraints on the language for modeling contracts.

Closure under limits models that any trace can be indefinitely extended with trace fragments, while the result is also a trace. This constraint is guaranteed by default by a category of TIOA: *Lemma 4.20* from [108] proves that an automaton with *finite internal non-determinism* has its set of traces limit-closed.

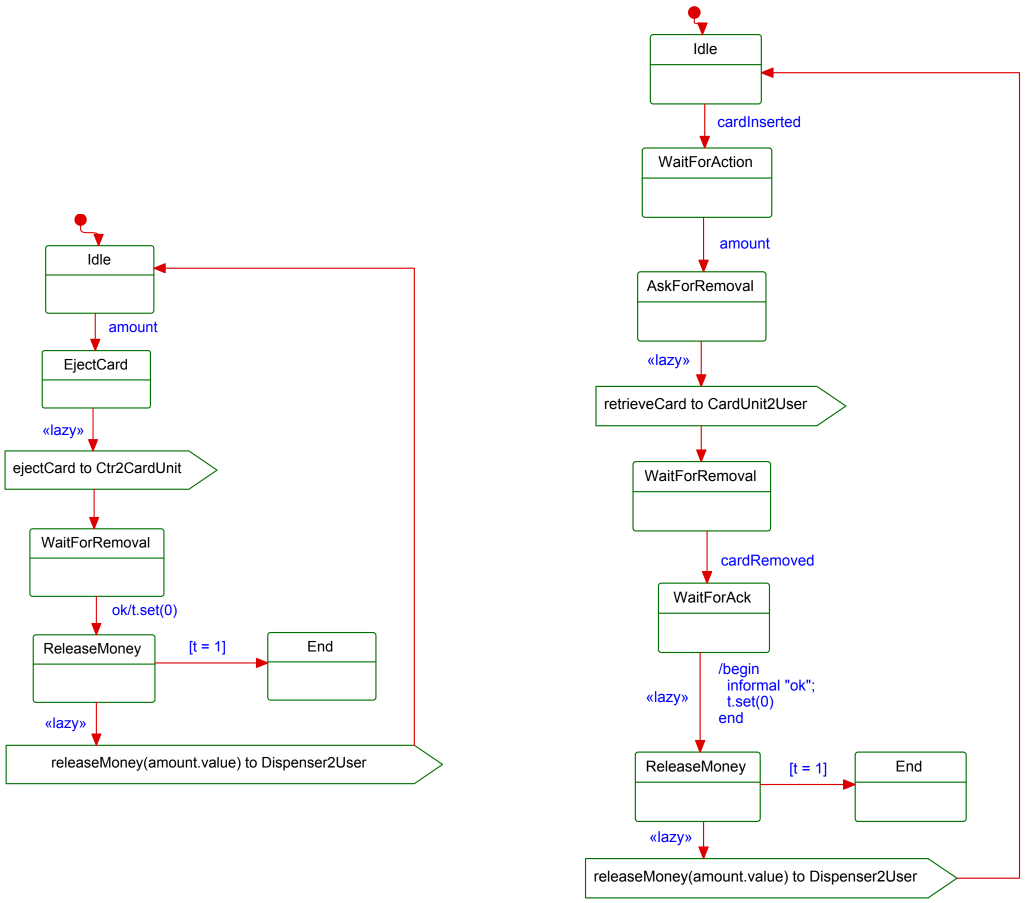
Definition 6.12 (Finite internal non-determinism). An automaton has finite *internal non-determinism* if:

1. the set of start states θ is finite and
2. $\forall \mathrm{x} \in Q$, $\forall \beta \in \mathit{tracefrags}(\mathrm{x})$, the set $\{\alpha.\mathit{lval} | \alpha \in \mathit{frags}(\mathrm{x}), \alpha \text{ \textit{has the last trajectory closed and traces}}(\alpha) = \beta\}$ is finite.

The first condition is satisfied by definition by our TIOAs which model only one initial state corresponding to the initial state of the state machine it represents. The second condition implies that, for any trace, an infinite set of execution fragments does not exist. In order for this condition to hold, cycles of silent transitions cannot be modeled in the state machine. By silent transition we mean a transition defined for an internal action except signal handling. A cycle of silent transitions might modify the values of variables thus leading to an infinite set of last states. Secondly, the timed semantics for all internal transitions is *eager*. Indeed, a lazy semantics would generate an infinitely-branching non-determinism, since the transition can be executed at any moment during time elapse and so change the state of the system. Defining transitions with internal actions as eager and forbidding the modeling of cycles of silent transitions ensures that the automaton is non-Zeno, i.e. does not execute an infinity of internal actions in a finite time. These conditions are sufficient to ensure finite internal non-determinism for an automaton.

Closure under time-extension lets time elapse in any state of the automaton. The easiest way to achieve time progress in a state machine is to stereotype outgoing transitions from the control state as *lazy*. This stereotype can be applied on transitions that perform an output, since internal transitions are already stereotyped with eager for closure under limits. Recall that signal consumptions are modeled as internal transitions, hence they have an eager semantics.

Indeed, this setting of urgency is sufficient to ensure closure under time-extension for a state machine that is non-Zeno. This is due to the fact that eager transitions are executed as soon as they are enabled, thus eventually leading to a state where either an output may occur (and the transition executing the output being lazy lets time progress to infinity) or to a final user-defined state, i.e. termination state without outgoing transitions, where again time may progress to infinity.



(a) Time-bounded behavior for *G_ Controller*

(b) Time-bounded behavior for *G_ sATM*

Figure 6.4 – Time-bounded behaviors for sATM example guarantees.

To summarize, the following stereotypes apply on TIOA transitions: (1) transitions with an output are lazy and (2) internal transitions are eager. Visible actions have a lazy semantics obtained from the output action at composition. Compound transitions are broken into a sequence of transitions with intermediate generated states which are evaluated in the order they are modeled.

We remark that this restriction does not allow specifying, in a guarantee, a hard upper bound for when an output/visible action may occur. However, a bound may be specified using a timed guard, but the interpretation is that either the output/visible action occurs within/after this bound or does not occur at all. Therefore, the execution of an output/visible action cannot be enforced and, moreover, cannot be performed at a given deadline. This is a limitation of the expressiveness of guarantees in our theory.

For the sATM running example, the guarantee of the *controller* depicted in Figure 6.1(b) is loose: it models that eventually the controller will release the money — it includes the case that this can never happen. We can strengthen this guarantee by modeling that if the amount is released then the action will take place in at most 1 time unit, while satisfying the closure under limits and under time-extension conditions. This new guarantee is represented in Figure 6.4(a): from the state *ReleaseMoney* there are two outgoing transitions. One transition eventually executes the *releaseMoney* output since it is lazy. The second transition models an internal action ε when the clock t is equal to 1 and moves to a final state *End*. Then either *releaseMoney* is executed in at most 1 time unit (including the 1 time unit moment) or never due to the evolution to the state *End*. Its modification imposes also the change of the *atm*'s guarantee to take into account the delay, modeled in Figure 6.4(b).

We remark that this new set of contracts can be used to verify a second requirement of the *atm*: the *atm* either releases the amount within at most one second after its controller is aware of the card removal or never.

6.5 Automatic Verification of Generated Proof Obligations

The contract theory we defined is based on the trace inclusion relation. However this relation is undecidable in the general case and cannot be automatically verified by tools except for restricted categories of timed automata [131, 155]. Two solutions can be envisioned for the automatical verification of refinement under context: (1) either by making additional hypotheses on the form of the abstract component and requiring it to be a *deterministic* timed safety property, which allows one to use *reachability analysis* for guaranteeing trace inclusion, or (2) by using timed simulation [154] which also guarantees trace inclusion.

We chose to use the first option based on model-checking, by applying reachability analysis on our TIOA models composed with a *timed property automaton* that models the requirement to be verified. This model-checking algorithm is implemented in the IF toolset [34] which serves to automatically verify our models. In the following we describe our technique to model-check contract satisfaction relations by deriving from the deterministic timed safety property modeled by a guarantee a timed property automaton and we show that this transformation is sufficient to satisfy trace inclusion.

A *timed property automaton*, introduced in Section 1.1, consists in a complete definition of a safety requirement: it defines an *"error"* state π to which incorrect behaviors will lead and synchronizes with the component under study $\mathcal{C}$ on common actions. The reasoning for proving contract satisfaction proceeds as follows: (1) transform the guarantee into a timed property automaton and (2) apply model-checking, i.e. run in parallel the component $\mathcal{C}$ and the property automaton and explore the final state graph to check if the error state π has been reached. Reaching the error state π signifies the violation of the contract satisfaction relation.

We start by defining the transformation process from a deterministic safety property to a timed property automaton. The mechanism is similar to the one defined in [41], albeit for untimed systems, and later used for automated assume-guarantee reasoning and the LTSA tool [84, 30].

Definition 6.13 (Timed property automaton). Given a *deterministic* TIOA $\mathcal{A} = (X_\mathcal{A}, Clk_\mathcal{A}, Q_\mathcal{A}, \theta_\mathcal{A}, I_\mathcal{A}, O_\mathcal{A}, V_\mathcal{A}, H_\mathcal{A}, \mathcal{D}_\mathcal{A}, \mathcal{T}_\mathcal{A})$, the *timed property automaton* for $\mathcal{A}$ is defined as the following TIOA $\mathcal{O}_\mathcal{A} = (X_\mathcal{A}, Clk_\mathcal{A}, Q, \theta_\mathcal{A}, \emptyset, \emptyset, V, H_\mathcal{A}, \mathcal{D}, \mathcal{T}_\mathcal{A})$ where:

- $Q = Q_\mathcal{A} \cup \{\pi\}$, where π is an additional error state,
- $V = I_\mathcal{A} \cup O_\mathcal{A} \cup V_\mathcal{A}$,
- $\mathcal{D} = \mathcal{D}_\mathcal{A} \cup \{(\mathrm{x}, a, \pi) | \mathrm{x} \in Q_\mathcal{A}, a \in V$ such that $(\nexists\, \mathrm{x}'.(\mathrm{x}, a, \mathrm{x}') \in \mathcal{D}_\mathcal{A}) \wedge (\nexists\, \varepsilon \in H_\mathcal{A} \wedge \mathrm{x}' \in Q_\mathcal{A}.(\mathrm{x}, \varepsilon, \mathrm{x}') \in \mathcal{D}_\mathcal{A})\}$.

The idea behind this transformation is that sequences of actions that are not explicitly modeled should be considered as erroneous behaviors. Since a timed property automaton is used to monitor a closed component, we consider the signature of $\mathcal{O}_\mathcal{A}$ to contain only visible actions, corresponding to the inputs, outputs and visible actions of $\mathcal{A}$. Then in every state of the automaton from which there is no outgoing internal transition, we complement the set of transitions with those "missing": for each visible action there must be a discrete transition either leading to a state defined in $\mathcal{A}$ or to π. So, the actions leading to π model the discrete actions that are not allowed to occur in a given timed sequence of $\mathcal{A}$.

We remark that the definition of the timed property automaton is similar to the modeling of requirements with observers in OMEGA. Indeed, an observer formalizes a safety property by already modeling the error state π via the error stereotype and defines only the set of visible actions, i.e. no inputs or outputs. Therefore, the same verification mechanism can be applied for checking the satisfaction of conformance. In this case π is not added to the $\mathcal{O}_\varphi$ if it has been modeled by the user.

However, for this method to work the component $\mathcal{A}$ must be a *deterministic* safety property both for visible actions and for internal actions. For internal actions, determinism means that there is at most one outgoing transition from a state. The guarantee is already required to have finite internal non-determinism. Therefore, in order to obtain determinism, we restrict the second condition of Definition 6.12 such that the cardinal of the set of last states to be equal to 2: the last state can be either the initial state if no internal transition has been fired or the state obtained by firing the transition. We remark that these conditions have to hold in the TIOA framework. It implies that in a SysML state machine, one is still able to model several outgoing internal transitions given that they are not to be enabled at the same time, e.g. two transitions handling different inputs or two transitions having disjoint guards.

The synchronization at run-time between $\mathcal{C}$ and the timed property automaton $\mathcal{O}_{\mathcal{A}}$ is defined by the following composition operator, denoted $\bowtie$. It is similar with the previous parallel composition operator described in Definition 6.5 with synchronization on the common visible actions and interleaving of the others. The operator can be applied on two timed input/output automata if they do not share any internal actions by label and they do not exhibit any inputs/outputs. The latter condition is motivated by the fact that the timed property automaton monitors a closed environment.

Definition 6.14 (Observer composition). Let $\mathcal{A}_1$ be a closed component and $\mathcal{A}_2$ a timed property automaton such that $E_{\mathcal{A}_1} \subseteq E_{\mathcal{A}_2}$. Then $\mathcal{A}_1 \bowtie \mathcal{A}_2 = \mathcal{A}_1 \parallel \mathcal{A}_2$ where the compatibility condition is relaxed to the constraint $H_{\mathcal{A}_1} \cap H_{\mathcal{A}_2} = \emptyset$.

The following result expressed on reachability analysis concludes the reasoning. Informally, not reaching the error state during any run of the composed component satisfies the trace inclusion relation.

Theorem 6.6. *If K_2 is a deterministic safety property and $reach((K_1 \parallel Env \parallel Env') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} = \emptyset$ then $K_1 \sqsubseteq_{Env} K_2$.*

Proof. This proof is realized by contradiction. We suppose that $K_1 \not\sqsubseteq_{Env} K_2$.
$\Longrightarrow \exists \sigma \in traces_{K_1 \parallel Env \parallel Env'} \wedge \sigma \notin traces_{K_2 \parallel Env \parallel Env' \parallel K'}$

Let $\sigma' a$ be a prefix of σ such that $\sigma' \in traces_{K_1 \parallel Env \parallel Env'} \cap traces_{K_2 \parallel Env \parallel Env' \parallel K'}$ and $\sigma' a \notin traces_{K_2 \parallel Env \parallel Env' \parallel K'}$, where a is a visible action. Such prefix exists because K_2 is a safety property.

Then $reach((K_1 \parallel Env \parallel Env') \bowtie \mathcal{O}_{K_2})(\sigma') = \{(q_1, q_2)\}$.

Concatenating a we obtain:

$$(q_1, q_2) \xrightarrow{a}_{(K_1 \| Env \| Env') \bowtie \mathcal{O}_{K_2}} \pi \implies$$

$\implies$ *reach*$((K_1 \parallel Env \parallel Env') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} \neq \emptyset$ in contradiction with the hypothesis. □

With this notation, the conformance step is written: *reach*$((A \parallel G) \bowtie \mathcal{O}_{\varphi}) \cap \{\pi\} = \emptyset$.

The fact that the abstract TIOA has to be deterministic is a limitation of this verification method which has an impact on the modeling of contracts since it can induce an overhead for their definition. The limitation is usually not problematic for verifying *contract satisfaction* as safety guarantees have to be expressed as time- and limit-closed TIOA and they can often be determinized. However, in order to establish dominance, one has to verify also "mirror" contract satisfaction, which is more tricky since until now we did not require assumptions to be safety properties. In consequence, modeling assumption as deterministic safety properties becomes necessary for using model-checking in combination with timed property automata on all proof obligations.

The sATM running example is in this case: the assumptions modeled need to be customized in order to describe safety properties. In the case of *A_ Controller*, the only modification to be made is to stereotype transitions with an output as lazy. However, the case of *A_ CardUnit* and *A_ sATM* is more challenging: the assumption expressed by Requirement 4.1 about the environment — the card must be removed within 5 time units — contains a hard upper bound and cannot be modeled by a safety property. A solution is to make use in the assumptions of the concrete environment *user*, possibly with a loose behavior with respect to card removal that is a delayable transition (as modeled by *A_ sATM*) denoted *user'*. In this case *A_ CardUnit* can be defined as the composition $user' \parallel G'\_Controller$, where *G'_ Controller* has the same signature as *Controller* in order for the refinement relations to hold. The signatures of *G_ sATM* and *G_ CardUnit* will also be identical to the one of their corresponding components. In consequence, *G'_ sATM*, *G'_ Controller* and *G'_ CardUnit* are merely identical to the components, except the *nok* branch which can be removed from the guarantees. This running example shows the important overhead modeling verifiable contracts entails with respect to small systems.

In consequence, if the assumptions cannot be described using deterministic safety properties, there are two solutions which may open up the possibility of automatically verifying proof obligations: either verify timed simulation which implies the satisfaction of trace inclusion or use, if possible, the concrete environment as assumption such that “mirror” contract satisfaction relations become trivial to verify. We will illustrate how the second option concretely works on the case studies presented in Part III.

6.6 Comparison with Related Approaches

As described in Section 3.1.2, there are several contract-based frameworks available for timed systems that are based on assume-guarantee reasoning. The meta-theory of [142] is the only one which proposes circular reasoning that allows to reduce dominance to a set of contract satisfaction proofs. To the best of our knowledge, this is the first instantiation of the meta-theory defined in [142] for timed systems with asynchronous communication.

The contract-framework proposed in [63] for the TIOA framework presented in [61, 62] defines contract satisfaction as timed alternating simulation while computing quotient. Formally, the relation is written $K \leqslant (A \parallel G)\backslash\backslash A$, where $\leqslant$ denotes simulation and $\backslash\backslash$ the quotient operator. This theory does not allow for refinement of signature between specifications. Moreover, quotient operator is partial: the conditions in which the operator can be applied and what happens if the result cannot be computed are not discussed. As an instantiation of the meta-theory from [17], dominance can be established only by composing contracts and verifying pairwise refinement between the composition and the abstract component. Finally, this framework does not clearly state which type of requirements can be verified nor how a requirement should be modeled. In consequence, the conformance step is not formalized. With respect to the target TIOA frameworks, a comparison is provided in Section 1.1.4.

In [44], a specification theory is developed for the TA of [7] with input/output distinction and which are extended with the notion of co-invariant on states in order to express liveness timing assumptions. This framework is suited to verify safety and bounded liveness properties (on finite traces). The refinement relation is identical to ours since it consists in timed trace inclusion, however the signature refinement is not considered here whereas it is explicitly handled in our framework. A second difference can be remarked with respect to the definition of contract: the framework from [44] defines an interface theory where a specification encompasses both the assumption and the guarantee. The advantage of having disjoint assumptions and guarantees is discussed in Section 3.1.2.

The theory from [43, 39, 42] covers these differences by proposing an assume-guarantee reasoning framework but only for untimed systems. Therefore, a contract is given by two sets of traces — one for the assumption and one for the guarantee, while a covariant inclusion relation on inputs and a contravariant relation on outputs is considered for contract satisfaction. Again, the refinement relation is given by trace inclusion. Similarly to the meta-theory from [17], checking dominance requires computing contract composition and afterwards verifying refinement. In contrast, our sufficient condition for dominance allows to perform verification on smaller components, while, in case of a violation, the erroneous contract could be more easily identified.

The work from [84, 85, 133, 54] focuses on the automatic generation of assumptions via a CEGAR approach or automatic learning such that assume-guarantee reasoning holds. One inconvenient is that a contract framework is not clearly defined: assume-guarantee reasoning is independently used from the notion of contract in order to establish compositional verification of requirement satisfaction. The implementation of this approach is provided in LTSA tool[2] for systems described by LTS, thus untimed. Yet, our verification method is greatly inspired from their verification strategy with respect to the transformation of a component into a property automaton.

We consider that one asset of our contract framework is that almost all proof obligations reduce to contract satisfaction checks and on which we can perform error diagnosis. Hence, our framework facilitates finding bugs at the different levels of the proof tree by clearly identifying the cause and possibly correcting it. The diagnosis strategy, which was used of the sATM running example, is discussed in the next chapter and it represents one of the important perspectives of our work.

6.7 Conclusion

In this chapter we defined our contract framework for system designs modeled with SysML as an instance of the meta-theory of [142], where the component framework is formalized by a variant of the Timed Input/Output Automata framework from [108] and the proof obligations each refinement relation generates are expressed as a timed trace inclusion relation. We showed that the reasoning with contracts is sound in our framework by proving that the compositionality properties required by the meta-theory hold. The contract-related notions and application methodology of the contract framework are illustrated with the sATM running example.

[2] http://www.doc.ic.ac.uk/ltsa/

In order to fully comply to the semantics of SysML components, two modifications were required to be made on the TIOA framework of [108]: (1) extending the set of actions with visible ones, which denote an input/output matching and which play an important role for behavior decomposition on several components and for proving refinement, and (2) restricting the clock rate to a derivative equal to 1, thus having a time expressiveness equivalent to the TA of [7]. However, this restriction does not impact the compositionality results required by the meta-theory, i.e. it is not used as hypothesis in the corresponding proofs. Therefore, our contract framework holds also for general hybrid systems having different clock rates.

Yet, this restriction opens up the possibility of automatically verifying refinement relations. We have chosen as conformance relation (on which refinement under context is based) the timed trace inclusion, for which we know that is undecidable in the general case of TA [7]. Proposition 1.1, which can be extended to timed words, gives a sufficient condition for satisfying trace inclusion by proving that simulation holds. Simulation is decidable for the TA of [7], which are time equivalent to our variant, but not for the the TA of [108].

We preferred a second solution for verifying trace inclusion: we use model-checking as automatic verification algorithm based on a transformation from component to requirement formalization and the application of reachability analysis for undesirable states. Therefore, the abstract component playing the role of the guarantee is represented by a timed property automaton which is synchronized with the component under study in order to address its satisfaction. The IF model-checker implements this algorithm and allows us to automatically verify the obtained proof obligations.

Our contract framework imposes some restrictions on what contracts can model. The formal theory holds if and only if the contract's guarantee is a safety property that does not describe hard upper time bounds for actions, i.e. a specific deadline when the action has to be performed. Moreover, the automatic verification can be applied only if the safety component is deterministic with respect to external actions and internal actions disjointedly. Therefore, both the assumption and the guarantee need to be designed as deterministic safety properties. The case of the assumption may be problematic for critical real-time system: the modeled environment for a component, which is constituted by other components, often exhibits hard clock bounds on actions. A solution would be to use the environment as assumption if it is not subject to combinatorial explosion such that the concerned verification steps become trivial, or to verify timed simulation on the obtained conformance relation

7 Implementation in the IFx2 Toolset

The last step for using the formal contract-based framework for requirement verification in system designs with SysML consists in bridging the gap between the two frameworks by proposing a model transformation from SysML to our variant of Timed Input/Output Automata and implementing it into a compiler. In this chapter we present the IFx2 toolset, which allows to model-check and simulate a SysML model extended with contracts respecting the OMEGA convention notation, and how it can be used to diagnose errors in the system model.

In Section 7.1 we sketch the mapping of a system model extended with contracts into a network of Timed Input/Output Automata and how the model can be explored in order to generate the proof obligations corresponding to the modeled refinement relations. Section 7.2 presents the compiler included in the IFx2 distribution that implements this transformation. Finally, Section 7.3 describes how error diagnosis can be performed for a system model with contracts.

7.1 Compiling OMEGA Designs with Contracts to TIOA

The transformation of a component-based system extended with contracts and modeled with the SysML notions presented in Sections 4.1 and 4.2 into a network of Timed Input/Output Automata (TIOA) ready for formal verification is realized in two steps: first transform each component into a TIOA and secondly generate the required proof obligations based on the refinement relations modeled in the design. Since a contract's assumption/guarantee is described by the same component language as the system, the same process for transforming them into TIOA is applied here.

7.1.1 Mapping Components into TIOA

Transforming a component into a TIOA is relatively straightforward. In the following we sketch a set of rules that allows this transformation. Our mapping follows the same strategy that has been described in previous work like [122, 109] or lately [128].

We consider that for each atomic component K of the extended system a timed input/output automaton $\mathcal{A}_K$ is generated. The reason for mapping directly components is the lack of a mechanism in the described TIOA formalism to define structured types and instantiation operations. The set of clocks $Clk_{\mathcal{A}_K}$ consists in all the attributes defined by the component of type *Timer*, while all the other modeled attributes form the set of discrete variables $X_{\mathcal{A}_K}$. It is assumed that each automaton $\mathcal{A}_K$ contains two implicit discrete variables: *queue* stores all incoming requests and dispatches them to be handled by the automaton and *location* models the current control state of the component. The valuation function for the location variable ranges in a finite domain as it is modeled by the component's state machine. At the level of the component's type, one may model relations: associations are handled based on their end elements that represent attributes in the corresponding classes, while generalization between classes is flattened and all inherited attributes and association ends are duplicated in the automaton corresponding to the child class instance.

The set of states of the automaton $\mathcal{A}_K$ is given by the valuation of all variables, where the initial state $\theta_{\mathcal{A}_K}$ either contains a user-defined value at initialization for variables or a predefined one like 0 for clocks or $\emptyset$ for the message queue.

The behavior of the component is modeled by its type's state machine that describes the transitions and trajectories of the automaton. A state machine transition is defined between a control source state s and a control target state s' on which we can evaluate a *guard* and execute several *effects*. A transition is usually enabled by a *trigger* or time delay deadline. Thus for each state machine transition a set of TIOA transitions is generated between two states q and q' where $q.location = s$ and $q'.location = s'$. We denote by $q.x$ the value of the variable x in the TIOA state q.

The guard models the conditions for which the TIOA transition exists given that it is satisfied in the starting state q, otherwise no transition is generated. In each state of the automaton there is a predefined transition for each input action a. Its effect is to add the signal to the queue, i.e. $q'.queue = [q.queue; a]$. Then a trigger m is transformed into a transition executing an internal action $\downarrow m$ that consumes the message m thus $s.queue = [m; a]$ and $s'.queue = [a]$. The declaration of an internal variable *queue* and the difference between an input, which adds a request to the queue, and an internal consumption action, which handles the top message from the queue, formalize the asynchronous communication between components.

The set of effects defined on a transition can consist in several signal outputs and assignments. For each effect an independent TIOA transition is generated. The signal sending action (or sendAction) becomes a transition with an output; the value of the location in the target TIOA state is either the target control state if there is only this effect modeled on the transition or an intermediate location is generated in case the effect is structured. This transition will synchronize with the input transition of the signal's target at composition and will modify the value of the queue. The assignment effect for discrete and clock variables is transformed into a TIOA transition with an internal ε action, which exists if and only if q' can be obtained from q by applying the assignment.

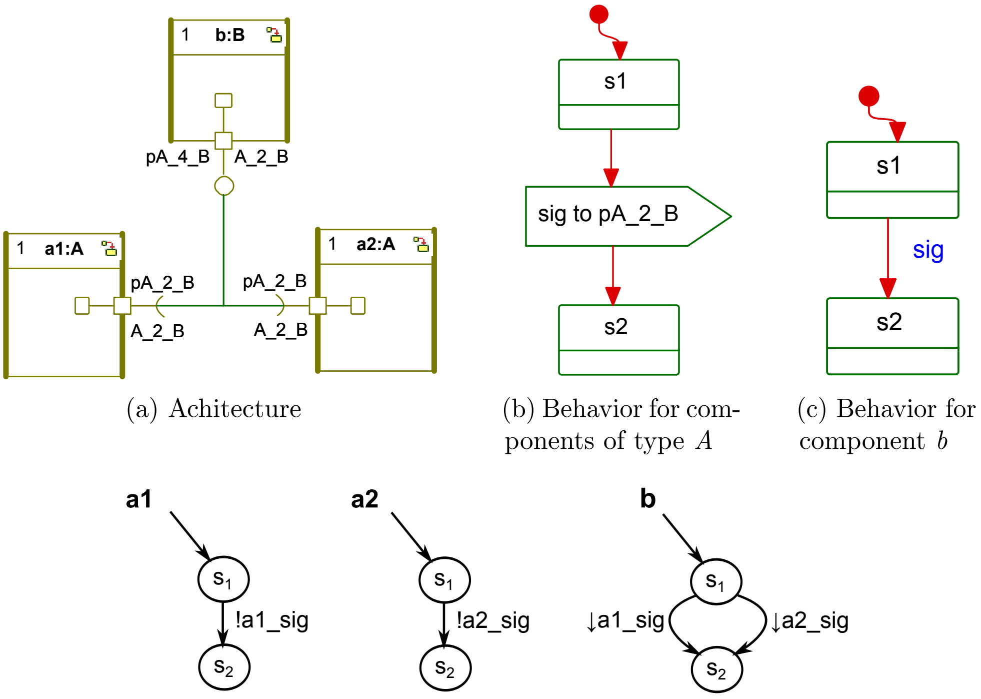
By default the time elapse in each state of the automaton is given by the set of all possible trajectories defined on $\{[0,t] | t \in \mathbb{R}_+\} \cup \{[0, \infty)\}$. This set of trajectories is controlled by the urgency labels of the outgoing transitions from $s = q.location$ in the state machine as follows:

- *lazy* does not add any restrictions;
- *eager* with no clock guard restricts the set of trajectories to point trajectory only, and
- *eager* with a clock guard restricts the set of trajectories so that they end in the smallest j where the guard is evaluated to true.

The formal definition of urgency stereotypes by trajectories is given in Section 1.1.4.

The set of signals that can be handled by the automaton is defined by the port types, while their input/output direction is given by the port directionality. So, all signal receptions modeled in the interfaces typing provided ports define the set of inputs $I_{\mathcal{A}_K}$ the automaton can handle and all signal receptions modeled in the interfaces typing required ports define the set of output actions $O_{\mathcal{A}_K}$ the automaton can perform. The set of visible actions for an automaton mapped from an atomic component is the empty set, $V_{\mathcal{A}_K} = \emptyset$. The set of internal actions $H_{\mathcal{A}_K}$ consists in instantiations of the silent transition ε, one for every transition defining a guard or trigger or assignment as obtained at transition mapping.

A particular attention must be brought to the name of signals since a model usually contains several components of the same type and they react to the same stimuli, in contradiction with the compatibility condition. Such an example is provided



(a) Achitecture

(b) Behavior for components of type *A*

(c) Behavior for component *b*

(d) Components behavior as TIOA

Figure 7.1 – An example for signal renaming in the SysML to TIOA transformation.

in Figure 7.1(a): two components *a1* and *a2* of the same type *A* communicate a signal *sig* to the component *b*. In this case, we need to make the difference between the instances of *sig* sent by *a1*, respectively *a2*. Our solution is to rename conflicting signals in the sender/receiver automata by appending their qualified name. Figure 7.1(d) presents the transformation of *a1* and *a2* into a TIOA, where *a1* sends the signal *a1_ sig* and *a2* the signal *a2_ sig*. In consequence, the two signals are disjoint. In order to represent the transformation we use the same notation convention as for LTS (see Section 1.1.1), where for compactness reason we represent a state only by its corresponding location in the state machine. If the receive automaton handles a signal that has multiple senders, the transition that handles the signal is duplicated for each sender: in Figure 7.1(d) the TIOA corresponding to the component *b* models two consumption ↓ transitions, one for each signal it receives. The target of a signal can be statically computed based on the traveled chain or ports and connectors. Since broadcast communication (i.e. one sender-multiple receivers) need to be explicitly modeled in a SysML design by different signals, there are no specific modifications to perform here, except the renaming the outgoing signals. This renaming convention actually flattens the TIOA network.

The case for a component representing a guarantee needs to be handled separately with respect to action renaming. The signals the guarantee executed need to be a subset (by name) of the signals that the component using it performs. Therefore, based on the *contractUse* relation we obtain for the guarantee the corresponding component and we rename the signals as they are instantiated in the component's TIOA representation. In the same time, the signals are renamed in the assumption of the contract such that they have a match in the guarantee and the component.

Finally, a composed component is not translated, since the TIOA framework does not define instantiation operations. The new signal names describe the hierarchical architecture of the system, while their renaming as both inputs and outputs based on the traveled chain of ports and connectors covers the entire architecture. As mentioned, the generated architecture of TIOA is flat. A hierarchical component is obtained at run-time via the composition operator.

The requirement formalization given by an observer in OMEGA is also transformed into a TIOA by applying the same rules. The difference consists in the fact that an observer does not have any inputs or outputs, all its actions being defined as visible. A transition is typed with a visible action if it is preceded by a send or informal match clause and with an internal action $\downarrow a$ if it is preceded by an acceptsignal clause. In each state the automaton defines a visible transition that adds the signal obtained from an input clause to the queue. The added signal can be later handled on the consumption transition. The timed semantics is defined as lazy for transitions resulting from send and informal match clauses and as eager for the rest. This timed semantics formalizes a safety property.

7.1.2 Generating Proof Obligations

Generating the relations that need to be verified for proving the satisfaction of a requirement is realized in the reversed order of the design steps, an in-depth exploration of the modeled contract tree. This is due to the fact that their definition relies on the inner structure of components which has to be computed, as it is the case for dominance. An algorithm is sketched in Listing 7.1.

It starts by computing the set of dependency relations modeled — *Conformance*, *Implementation* and *ContractUse* are type of *Dependency* — and the set of requirements that need to be verified with contracts. For each requirement, a set of proof obligations is generated.

First, we obtain via a *Conformance* relation the contract which conforms to the requirement. Because we require that each system design extended with contracts is complete, formalized in Listing 5.8, then we are sure this top contract exists. Then, we generate with the *generateConformanceProofObligation* function the conformance proof, i.e. $c \preceq o$ with the notations from Listing 7.1.

The next step consists in the verification that the "mirror" contract is satisfied by the environment of the component using the contract c for the requirement o, i.e. *generateMirrorContractSatisfactionProofObligation* function.

The third step consists of iterating through the modeled dominance relations as described by the function *generateRecursiveProofObligations*. This function is called in the context of the property using the contract c. Such an algorithm has been provided and explained in Listing 5.6 in the OCL language. There are two cases to be considered here: either the contract c is refined with respect to the requirement o or it is a contract used by a component that is not further decomposed. If the contract is refined then we compute the set of dominating contracts, as described in Section 5.1, and we generate with the *generateDominanceProofObligations* function the set of proofs that need to be verified. Next we verify if each of the dominating contracts is refined in the context in which is used via the recursive call of the proof obligation generation function.

If the contract is not refined, it means that we have reached the fourth step of the methodology and we generate a contract satisfaction proof, the *generateContractSatisfactionProofObligation* function. Recall that all operation calls are realized on the context of the component using the target contract.

Listing 7.1 An algorithm for generating proofs obligations from a system model extended with contracts

```
procedure generateRecursiveProofObligations (c:Contract, o:
  SafetyProperty, dependenciesList:List) is
  if isRefined(c, o) then
    refinementContractsList := getRefinementContractsOf(c, o,
      dependenciesList);
    generateDominanceProofObligations(c, refinementContractsList);
    for each c':Contract in refinementContractsList do
      generateRecursiveProofObligations(c', o, dependenciesList)
    end for
  else
     generateContractSatisfactionProofObligation(c);
  end if
end procedure
```

```
begin
  dependenciesList := getDependenciesFromModel();
  propertiesList := getSafetyPropertiesFromModel();

  for each o:SafetyProperty in propertiesList do
    contractConformingSafetyProperty := getContractConformingTo(o,
       dependenciesList);
    generateConformanceProofObligation(c, o);
    generateMirrorContractSatisfactionProofObligation(c, o);
    generateRecursiveProofObligations(c, o, dependenciesList);
  end for
end
```

7.2 Tool Architecture and Functionalities

The transformation previously presented is implemented in the IFx2 Toolset for OMEGA models, by updating the proprietary *uml2if* compiler that was compatible only with UML 1.3 models. Two tasks had to be performed: (1) add all modeling elements specific to UML 2.x (e.g. ports, connectors, composite structures) and ensure that the strong typing rules are satisfied and (2) implement the transformation to TIOA. Moreover, the interactive simulator of OMEGA models, contained in the IFx toolset, was updated for taking into account the new modeling elements.

With respect to the technological choices, *uml2if* is still a proprietary tool developed with Java, Eclipse UML 2.x and Eclipse EMF[1]. The usage of Eclipse UML 2.x/EMF is motivated by the fact that it offers a rich API fully compliant with the UML 2.x / SysML standards, which can be used from stand-alone Java applications, and offers a good basis for compatibility with UML/SysML modelers like IBM Rhapsody or Papyrus. In consequence, the transformation encoded by the compiler has been fully developed with Java. An overview of the *uml2if* compiler is given in the package diagram represented in Figure 7.2. Some metrics with respect to the Java code are presented in Figure 7.3. We remark that the tool contains almost 13,000 lines of code.

With respect to its features, the compiler takes as input a (system) model in the XMI 2.0 format and produces the IF textual representation of the network of TIOA. Its usage is described by the following command line:

```
uml2if [options] <filename.xmi>
```

[1] http://www.eclipse.org/modeling/emf/

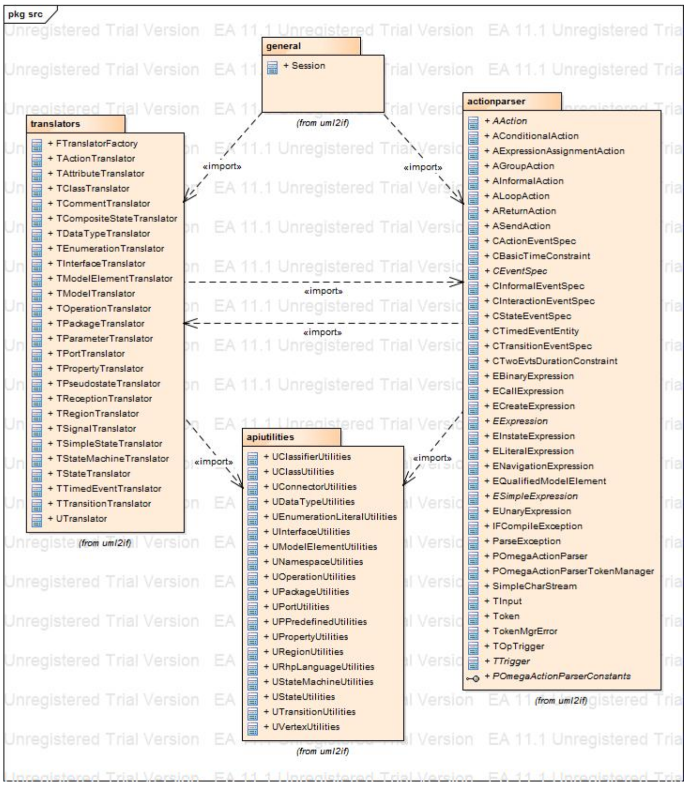
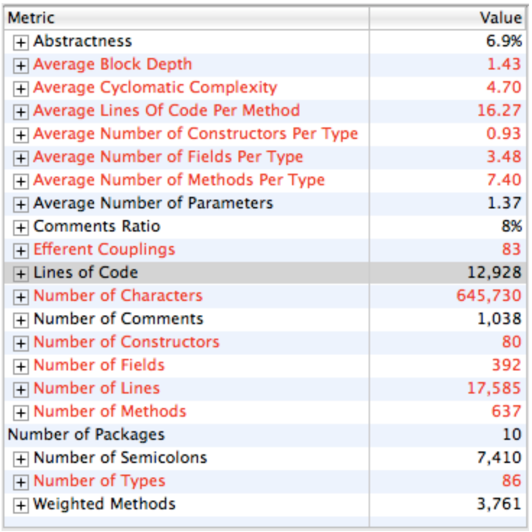


Figure 7.2 – *uml2if* package diagrams, with the classes each package defines.



Metric	Value
⊞ Abstractness	6.9%
⊞ Average Block Depth	1.43
⊞ Average Cyclomatic Complexity	4.70
⊞ Average Lines Of Code Per Method	16.27
⊞ Average Number of Constructors Per Type	0.93
⊞ Average Number of Fields Per Type	3.48
⊞ Average Number of Methods Per Type	7.40
⊞ Average Number of Parameters	1.37
⊞ Comments Ratio	8%
⊞ Efferent Couplings	83
⊞ Lines of Code	12,928
⊞ Number of Characters	645,730
⊞ Number of Comments	1,038
⊞ Number of Constructors	80
⊞ Number of Fields	392
⊞ Number of Lines	17,585
⊞ Number of Methods	637
Number of Packages	10
⊞ Number of Semicolons	7,410
⊞ Number of Types	86
⊞ Weighted Methods	3,761

Figure 7.3 – Some calculated metrics on the Java code of the *uml2if* compiler.

The following options are available:

- -*uml* handles a UML compliant model;
- -*sysml* handled a SysML compliant model. For example, if flowports are

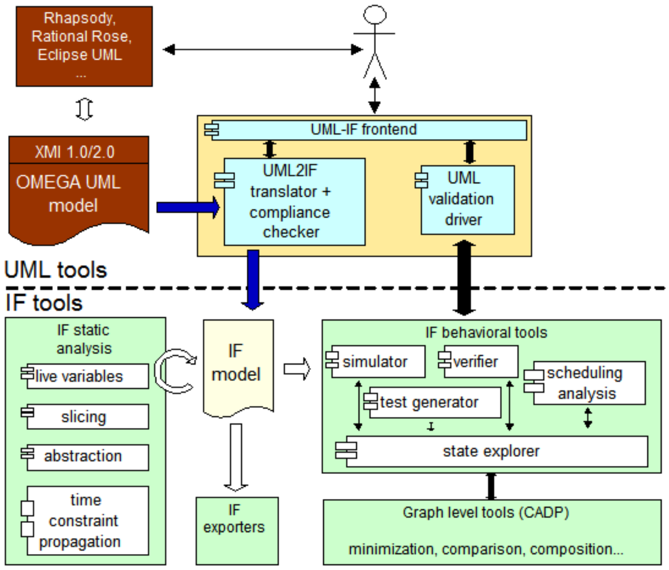


Figure 7.4 – The IFx2 Toolbox.

modeled a warning is produced by the compiler.

- *-rhapsody* handles a Rhapsody generated XMI file;
- *-rhplang* ensures the translation of a subset of the Rhapsody C language for actions;
- *-papyrus* handles a Papyrus model, which is already saved in the XMI format;
- *-sa* handles a particular version of Papyrus that saves association relations separately;
- *-eager* considers the eager-lazy time model, i.e. transitions that are not typed lazy are by default considered eager;

as well as others that are not discussed here. If errors are found in the model, the compiler produces an error message describing it and stops its execution. In consequence, the set of well-formedness rules OMEGA describes are guaranteed to be satisfied. The position of the compiler in the IFx2 Toolset is represented in Figure 7.4.

For example, on the sATM running example the following command line was used to generate the TIOA representation: `uml2if -sysml -rhapsody -eager satm.xmi`.

Finally, with respect to its functionality, the compiler implements the mapping described in Section 7.1.1 for system models extended with contracts. The algorithm presented in Section 7.1.2 for the generation of proof obligations is currently under development. We mention that this tool[2] has been assessed on industrial-grade system models, which we will see in Chapter 9.

7.3 Error Diagnosis for Contract-Based Reasoning

Within the proof obligations set, one or even several checks may not be satisfied. In this case we have to perform a diagnosis in order to establish if either the requirement is not satisfied or the set of contracts defined needs to be refined in order to prove the satisfaction of the requirement. We base this diagnosis on the generation of a counterexample which in our verification algorithm will lead to the error state π and use the same approach as for CEGAR [50]. This diagnosis is supported by the IFx2 toolset, which implements the verification method described in Section 6.5. The workflow of an OMEGA model through the IFx2 environment for both verification and diagnosis is showed in Figure 7.5.

We can distinguish two cases for which the solution depends on whether the reasoning has been applied for design or for verification: (1) a contract satisfaction or dominance verification fails or (2) the conformance verification fails. For the first case, if we are in a design approach and all previous steps have been proved correct, we have to refine the source component/contracts such that the counterexample is eliminated. This could possibly imply that the developed components are correct-by-construction with respect to the requirement. If we are in a verification approach with a completely modeled system, one should refine the target contract(s) since it is more frequent that the designed abstractions (in the form of contracts) are erroneous.

For the latter case, we have at first to verify on the concrete system if the generated counterexample is a spurious one due to the abstractions defined or it is a relevant one, which means that the system does not satisfy the requirement. Verifying if a counterexample is spurious or valid can be realized in our toolbox via interactive simulation. For a spurious counterexample, one should refine the top contract and re-verify at least the upper dominance step and the "mirror" contract satisfaction. Possibly, iterations of refinement of contracts must be performed until all checks pass. For a correct counterexample, the modeler should redesign the implementations (i.e. leaf components) on which the requirement is expressed. For this, the contract-based reasoning can be applied in a design approach in order to derive the correct contracts that satisfy the relations they are involved in, towards correct implementations.

[2] The *uml2if* compiler and *if2gui* simulator are available for download in the distribution of the IFx-OMEGA toolset at http://www.irit.fr/ifx.

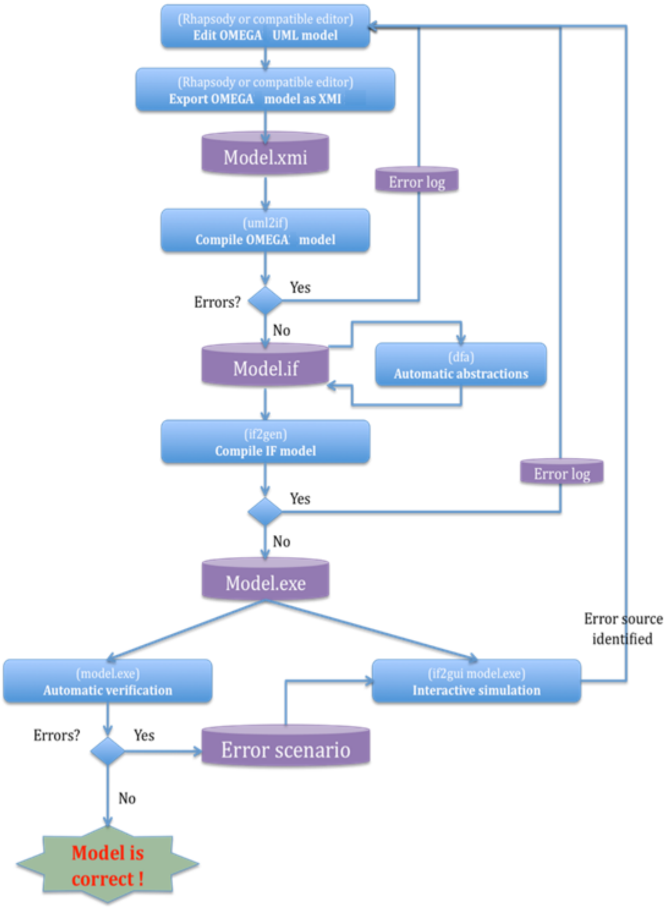


Figure 7.5 – IFx2 workflow for verifying and diagnosing a system design.

This diagnostic reasoning has been applied in Section 6.3 for modeling the correct contracts for the sATM running example. It allowed to detect the modeling errors that were presented.

We remark that, in the case of our tools, the generated counterexample is expressed on the TIOA level, while the refinement of contracts/components is realized in a high-level modeling language. The bridge between the two frameworks is unidirectional, since the transformation presented in Section 7.1 is given from SysML to TIOA. In order to exploit the error scenario, the developer has to apprehend in details the system and to make use of his experience for performing refinement. This point is an open question for which current research provides some options [56, 3] and it is outside the scope of this thesis.

7.4 Conclusion

In this chapter we presented a mapping of the component language (extended with contracts) into a network of Timed Input/Output Automata and a model exploring algorithm that allows to generate the proof obligation associated to each modeled refinement relation. The approach is implemented for the OMEGA working context in the IFx2 toolset by the *uml2if* compiler, where the generation of proof obligations is currently under development. This tool, which is available for download and evaluation, had to be updated for UML 2.x and SysML by taking into account new modeling elements and their well-formedness rules. The implementation of the mapping rules opens up the possibility to automatically verify and validate OMEGA models extended with contracts with the IF toolset. In case of requirement (including guarantee) violation, we have sketched a diagnosis process, which was applied on the sATM running example, and it constitutes one direction of our future work.

Experimental Results Part III

8 A Parametric Case Study for Comparing Verification Results

In this chapter we apply our contract-based framework on a parametric case study modeled with OMEGA, from contract specification to the verification of proof obligations. While the sATM running example was used to illustrate the theoretical concepts we defined, this case study aims to complement it by focusing on how the proposed verification method can be used to check the satisfaction of proof obligations. The ultimate goal is to provide a comparative efficiency study between monolithic model-checking and the contract-based model-checking we presented. Being a parametric design, i.e. internal variables are user-bounded, it also allows to assess the efficiency of the defined contract-based method for different state space sizes.

8.1 System Description and Contracts

The parametric case study is essentially made of two components $k1$ and $k2$ that use a timed protocol to coordinate in order to emit an alternating sequence of a's and b's. A nominal scenario of this system is presented in Figure 8.1: the component $k1$ sends to its environment e a signal a and to $k2$ a signal p. The component $k2$ receives p and sends a signal b to the same environment e. Next, $k2$ waits for 10 time units and sends q to $k1$ which restarts the behavior of the entire system.

In order to parameterize the system, we define two integer variables i and j for $k1$, respectively $k2$, which count the number of a's and b's sent to e. Moreover, for showing how the signature refinement works when contracts are defined and refinement is verified, we model a component $k3$ in Figure 8.3(c), which sporadically sends m and u to $k1$, respectively $k2$.

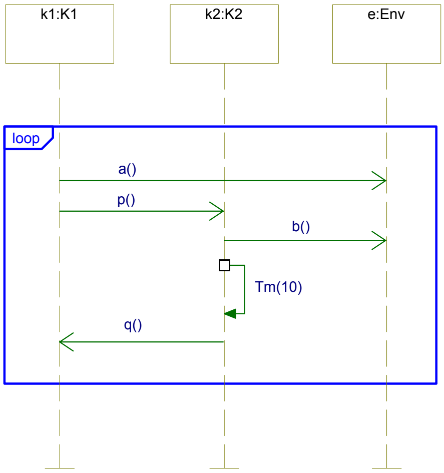


Figure 8.1 – A nominal scenario for the parametric case study.

The architecture of this system[1] is represented in Figure 8.2 and consists in two hierarchical layers: the subsystem K composed of $k1$, $k2$ and $k3$, and its environment Env. We describe now the communication protocol between $k1$ and $k2$ by taking into account $k3$: $k1$ sends a message a to the environment (port $pK1\_2\_Env$) and a message p to $k2$ (port $pK1\_2\_K2$), then awaits a message q from $k2$ (port $pK2\_2\_K1$). If q is received before the deadline set to the clock *clock*, $k1$ emits a again (the q-a cycle), otherwise it goes back to the initial state when q is received. In addition, $k1$ can answer to a message m (port $pK3\_2\_K1$) with a message $n(i)$ (port $pK1\_2\_K3$) in any state (the m-n cycle). The behavior for K_1 is represented in Figure 8.3(a).

The component $k2$ waits for p then sends a message b to the environment (port $pK2\_2\_Env$). After that, it waits for the *clock* deadline and sends q to $k1$; if a p request is received during this time, b is emitted again (the p-b cycle). Similarly, $k2$ can answer to a message u (port $pK3\_2\_K2$) with a message $v(j)$ (port $pK2\_2\_K3$) in any state (the u-v cycle). The state machine of K_2 is modeled in Figure 8.3(b).

Finally, the behavior for the environment component e is represented in Figure 8.3(d) and consists in consuming the messages received from the composed component k.

On this system, we are interested in verifying the following requirement:

Requirement 8.1. *The component k emits a sequence of alternating a's and b's.*

[1] *Notation.* The name of the ports starts with "p", followed by the name of the sender's type, followed by "2" and the name of the receiver's type. The name of interfaces uses the same convention where "p" is replaced with "I".

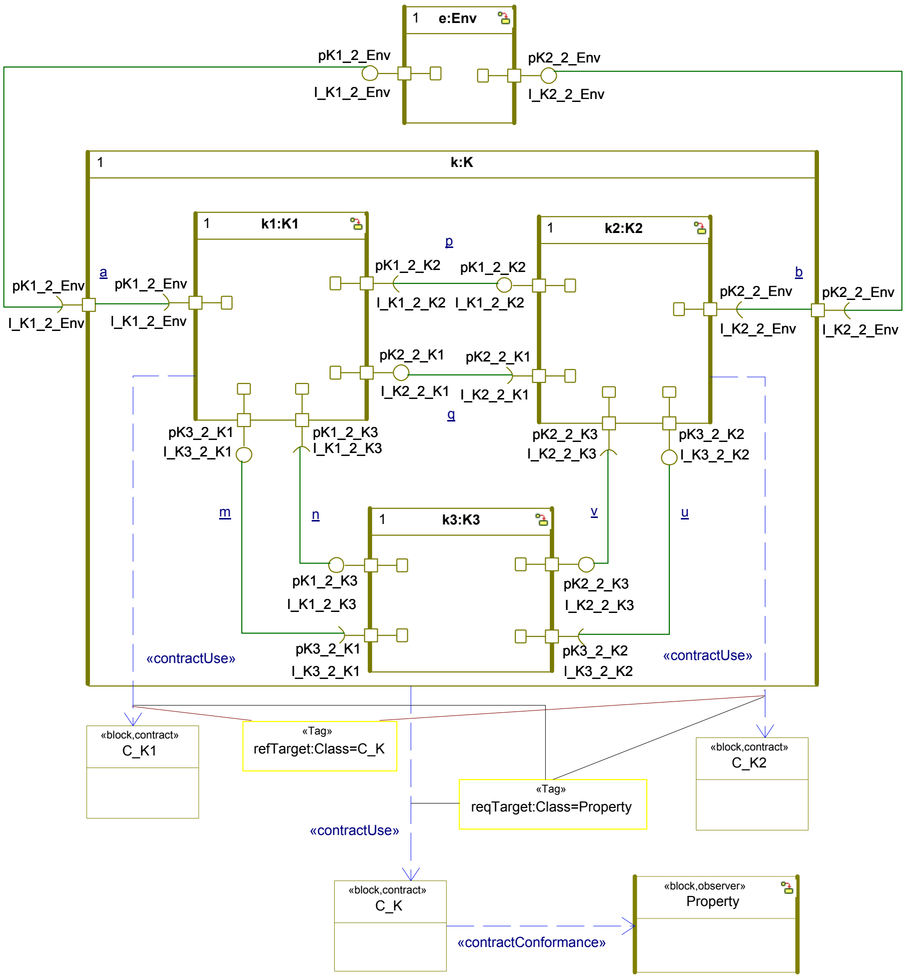
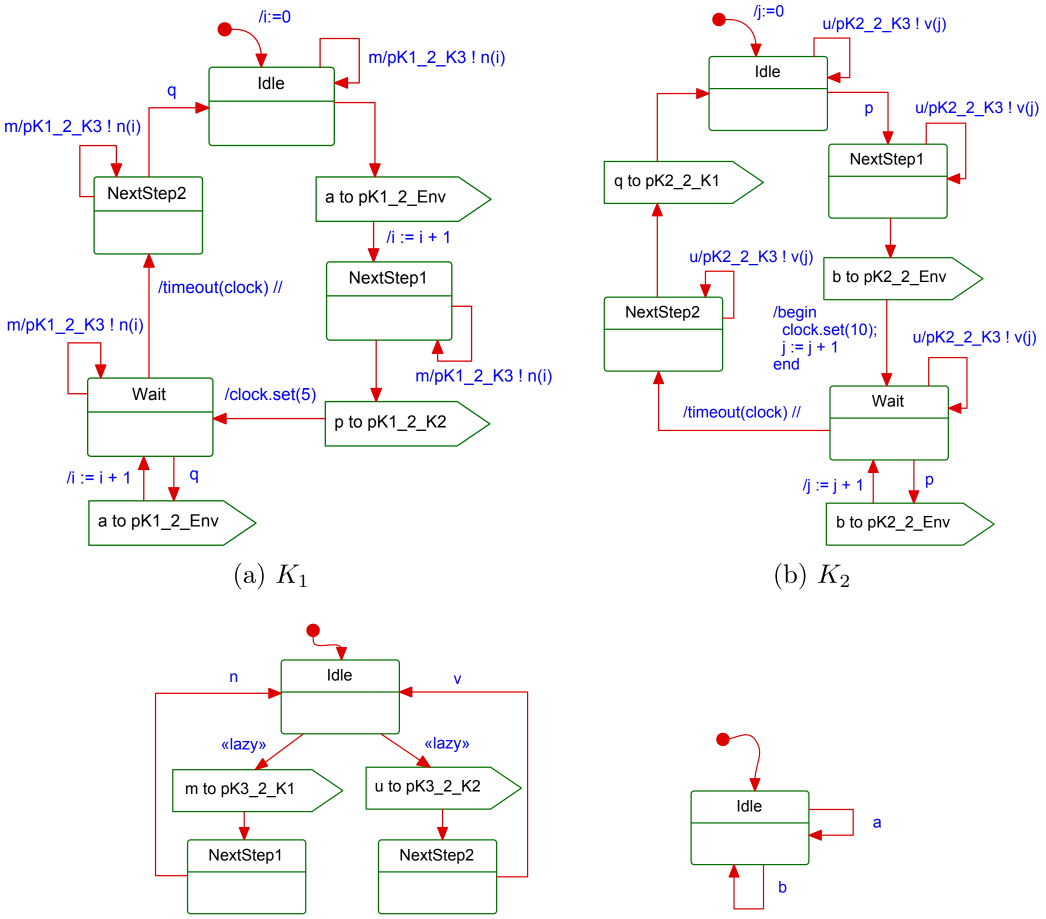


Figure 8.2 – Architecture of the parametric example K and its contract extension.

The requirement is formalized by the observer *Property* in Figure 8.4. The observer synchronizes with each execution of a and b and if detects any of the trace fragments $\{b, aa\}$ starting in the location *ObserveA*, then it moves to the *Error* state.

It is interesting to note that this requirement can also be parameterized. In the presented setting the deadline for the clocks modeled in $k1$ and $k2$ is respectively set to 5 and 10 time units. However we can consider general values as δ_1 for $k1$ and δ_2 for $k2$. The requirement is satisfied by the component k in the general case, if $\delta_1 < \delta_2$. In the following we do not consider this second parameter, but it can be easily explored.

As we will later see, the monolithic model-checking fails to produce a result for even medium bounds of i and j like 100. We set out to use the contract-based approach. We start by identifying the system under study as the composed component k



(a) K_1

(b) K_2

(c) K_3

(d) Env

Figure 8.3 – Behavior of the components involved in the parametric example.

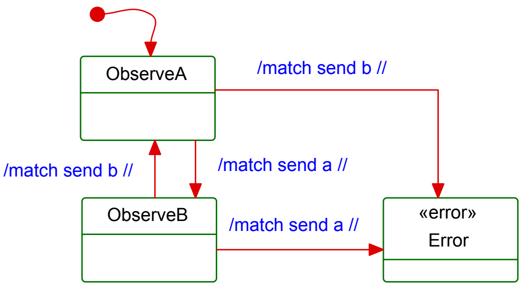
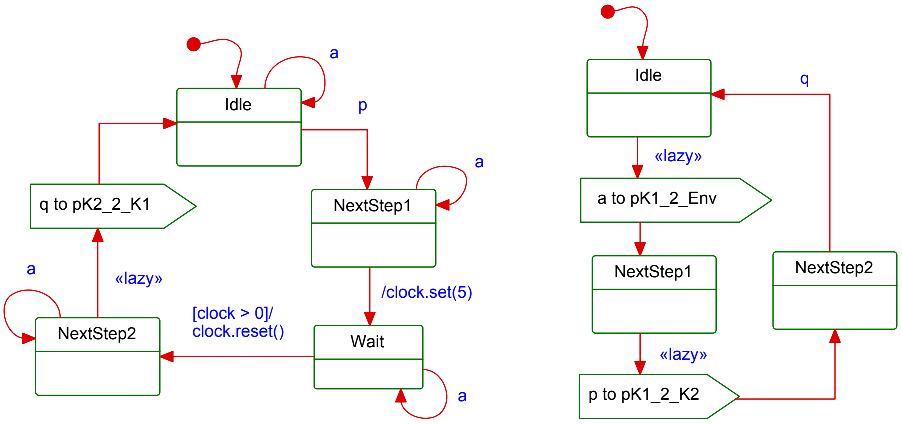


Figure 8.4 – Formalization of Requirement 8.1 by the observer *Property*.

and the environment as the component e. Based on the system's architecture, we have to model a set of contracts for the components $k1$, $k2$, $k3$ and a global contract for k and apply the dominance step once. Therefore, in Figure 8.2 we have represented the contracts together with the required *contractUse* relations tagged



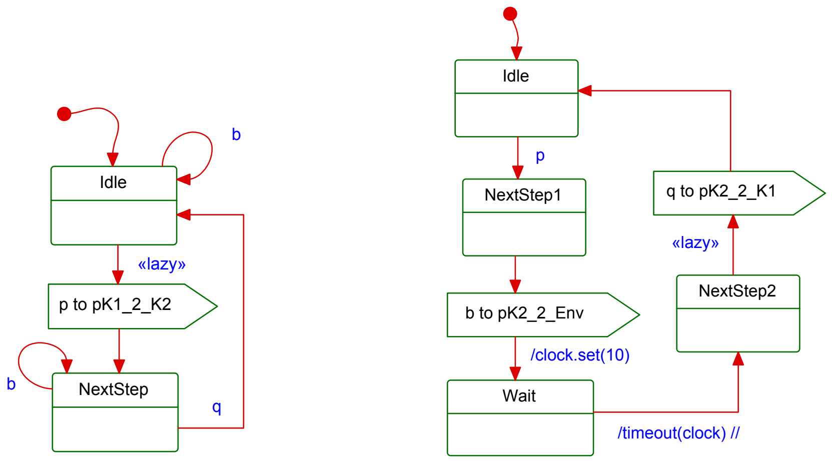
(a) Assumption $aK1$ behavior (b) Guarantee $gK1$ behavior

Figure 8.5 – Contract for the component $k1$.

with *Property* as requirement and C_K as refined contract for modeling dominance. The *contractConformance* relation completes the set of proof obligations that need to be verified.

Figure 8.5 presents a contract C_K1 for the component $k1$. The assumption is that the environment sends a q after at least 5 time units since a p is received and the component guarantees that consecutive a's are triggered by a message q from the environment. For $k2$, the contract C_K2 is represented in Figure 8.6. The environment guarantees that it will wait for a q between sending two consecutive p's and the component guarantees that it waits for a p before sending a b and then for a delay of 10 time units before a q. Since Requirement 8.1 is defined on the subset $\{a, b\}$ of actions, the component $k3$, whose signature is $\{m, n, u, v\}$, does not contribute to the satisfaction of this property. Its contract for this requirement is given by two empty timed input/output automata, i.e. the sets of variables and actions of the automaton are empty and all trajectories up to ∞ are admitted. Therefore, we do not require to model this contract and it is not represented in Figure 8.2.

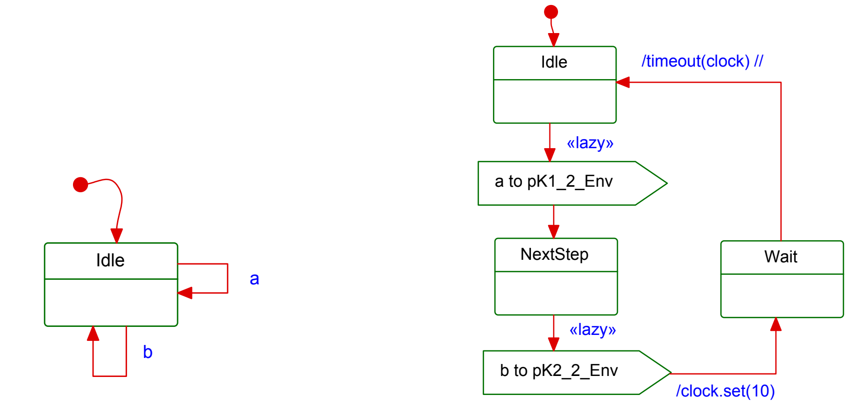
Figure 8.7 contains a top contract for the subsystem k. This contract guarantees that if an a message followed by a b message are sent to the environment then at least a delay of 10 time units will elapse between 2 cycles.



(a) Assumption $aK2$ behavior

(b) Guarantee $gK2$ behavior

Figure 8.6 – Contract for the component $k2$.



(a) Assumption aK behavior

(b) Guarantee gK behavior

Figure 8.7 – Top contract for the component k.

8.2 Contract-based Verification Results

The first verification step consists in checking contract satisfaction. This generates the following proof obligations:

(1.1) $k1 \sqsubseteq_{aK1} gK1$,
(1.2) $k2 \sqsubseteq_{aK2} gK2$ and
(1.3) $k3 \sqsubseteq_{aK3} gK3$.

Since $gK1$ and $gK2$ are deterministic safety properties we use the verification method described in Section 6.5. We present the transformation process from component to timed property automaton in OMEGA, thus by using an observer

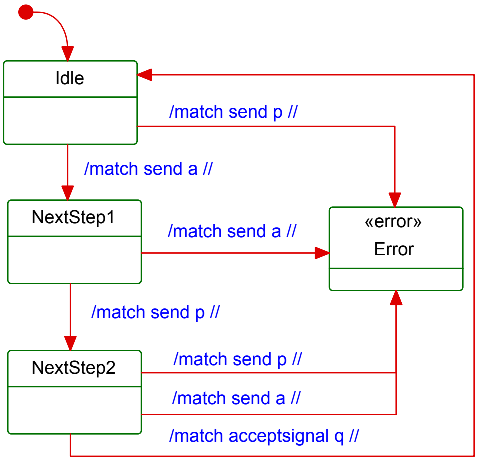


Figure 8.8 – Timed property automaton obtained from the component $gK1$ represented using an OMEGA observer.

to model the safety property. The obtained observer from $gK1$ is illustrated in Figure 8.8. In the location *Idle* it waits to synchronize with the action a. If instead a p is produced, then it moves to the location *Error*, otherwise it moves to *NextStep1*. It waits for a p action to be executed and to move to *NextStep2*, otherwise an error has occurred. In location *NextStep2*, the timed property automaton eventually waits to receive q and consumes it as soon as the action is enabled. During the wait, the component should not execute a or p represented by transitions to the *Error* location.

The system as it is described is infinite since the variables i and j and the queue length are not bounded and therefore model-checking cannot be applied. We bound in the following i and j with different limits, e.g. 5, 10 and 100, and the queue length to 1. We prove contract satisfaction for different bounds and the quantitative results obtained — number of explored states, transitions and verification time — are available in Table 8.1. We remark that the empty timed input/output automaton is the weakest component that refinement under context always satisfies and, in consequence, the verification is not performed.

The second step consists in proving that the set $\{C\_K1, C\_K2, C\_K3\}$ dominates C_K. For this we apply Theorem 6.5, since all components from contracts are timed safety properties, and we obtain the following proof obligations:

(2.1) $gK1 \parallel gK2 \sqsubseteq_{aK} gK$,
(2.2) $aK \parallel gK1 \sqsubseteq_{gK2} aK2$ and
(2.3) $aK \parallel gK2 \sqsubseteq_{gK1} aK1$.

We have dropped $aK3$ and $gK3$, since the empty component is the identity element for composition and it is always satisfied by refinement under context. Since $aK1$ and $aK2$ are deterministic safety properties we apply the same verification method for checking dominance. The obtained results are depicted in Table 8.1. It is interesting to observe that this verification step needs to be performed only once independently from the bound for variables i and j since they do not appear in the abstraction.

The third step which consists in checking the satisfaction of C_K^{-1} is trivial since e and aK are identical. The corresponding proof obligation is written: $e \parallel gK \parallel gK' \preceq e \parallel gK \parallel gK'$.

The fourth step in the verification of a system model is to prove that the top contract satisfies the global property, i.e. $A \parallel G \preceq \varphi$, which is true for this case study because the requirement does not posit any condition about the delay between cycles of a and b.

Table 8.1 presents some quantitative measures (number of transitions and of states explored and time needed for verification in seconds) for each verification step of the running example after bounding the counters i and j and the queue from the additional defined environment to the component[2]. The first column corresponds to the verification of the property on the whole system without contracts. It is interesting to note that the longest verification step with contracts is an order of magnitude smaller than the monolithic verification in this case, the explosion being caused by messages exchanged with $k3$ which are abstracted away from the contracts.

[2] We were using a IA64 computing server with 16GB of memory.

		Monolithic	Step 1			Step 2			Step 3	Step 4
			1.1	1.2	1.3	2.1	2.2	2.3		
max(i) = max(j) = 5	No. of states	41504	364	3372	-[3]	158	148	231	-[4]	51
	No. of transitions	79249	604	5070	-	239	229	422	-	65
	Time (sec)	6.86	0.1	0.42	-	0.04	0.04	0.03	-	0.02
max(i) = max(j) = 10	No. of states	400711	13422	10702	-	*[5]			-	*
	No. of transitions	827591	23210	15925	-				-	
	Time (sec)	78.84	1.51	1.21	-				-	
max(i) = max(j) = 100	No. of states	∞	1089102	809542	-	*			-	*
	No. of transitions	∞	1853540	1190290	-				-	
	Time (sec)	∞[6]	148.44	123.39	-				-	

Table 8.1 – Verification results for without/with the contract-based methodology on the parametric case study.

[3] Since an empty timed input/output automaton is the weakest component always satisfied, this verification step has not been performed.

[4] This verification is trivial since the environment is identical to the top assumption.

[5] The results are identical as for max(i)=max(j)=5

[6] The monolithic model-checking has been halted after 10 minutes without results.

8.3 Conclusion

The sATM running example on which we illustrated our concepts has revealed the overhead the modeling of contracts generates in contrast with monolithic model-checking. Indeed, for systems that generate a relatively small state space at execution, finding the correct contracts that allow to prove the satisfaction of requirements may be a more complex and costly task than directly applying model-checking.

The interest of applying contract-based reasoning is engendered by the generation of a larger state space which cannot be explored even with reduction techniques. This case study shows us that the complexity of defining contracts can be worth to consider contract-based verification even for small case studies. The quality of these verification results can be argued by the quantitative measures which allow to compare the two approaches. While contract-based model-checking allows to verify requirement satisfaction in a reasonable time for medium bound values for i and j, the monolithic version is not able to get through the state space. We mention here that partial order reduction has been applied on-the-fly when performing monolithic model-checking.

These positive results encourage the use of contract-based reasoning for requirement specification and satisfaction, especially for real-life systems that are often subject to state space explosion. We are not concerned here by finding a general threshold with respect to the system's complexity from which the performance of contract-based verification is greatly superior to other approaches and, so, making it recommendable for usage.

9 A Real-Life Case Study: The Automated Transfer Vehicle

This chapter presents the Solar Generation Wing Management System (SGS) of the Automated Transfer Vehicle (ATV) case study and its verification and validation with the contract-based reasoning technique. The ATV, developed by Airbus Defence and Space (ADS)[1] is a space cargo ship launched into orbit by the European Ariane 5 launcher with the aim of resupplying the International Space Station. The SGS system described here is responsible for the management of the solar arrays that provide the vehicle with the energy needed to fulfill its mission. It contains the functional chains that realize the solar arrays deployment and rotation.

9.1 System Description and Architecture

The system's design, illustrated in Figure 9.1, has been reverse engineered from the actual system by the ADS engineers, for the purpose of this case study. It is described with OMEGA SysML and the IBM Rhapsody tool. The model has a 4-layer architecture structured in a set of hardware and software entities that captures its timed behavior. Figure 9.1 adopts a high-level view of the main components, without the details of their substructure:

- The mission and vehicle management component (*MVM*) "simulates" a finite mission scenario going through the two operating modes of the SGS described below.
- The *SOFTWARE* component, subsystem of the Flight Application Software, consists of three sub-components, each with a specific function. They react to requests coming from the MVM and control the hardware by executing

[1] http://airbusdefenceandspace.com/

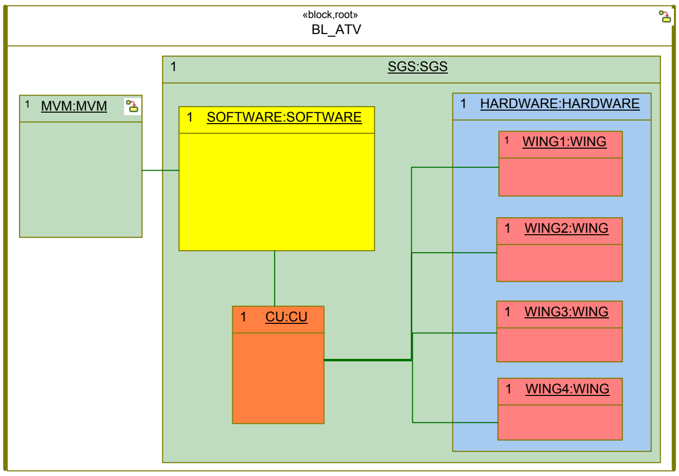


Figure 9.1 – An overview of the SGS model in Rhapsody SysML.

automated procedures in response to MVM demands.

- The *HARDWARE* component contains the four solar arrays of the ATV. This component has more than 70 pieces of equipment with multiple levels of redundancy for achieving reliability and availability in case of failures. Every wing is held in its initial position by four hold-down and release systems (HDRS). In order for the deployment to occur each HDRS has to be set loose. This is realized, for each wing, by eight thermal knives (TK); 2 TKs are needed for each HDRS — one for the nominal case and a redundant one in case of anomaly. Each time a HDRS is cut, a wing locking mechanism evolves to a deployed state for that array.
- The command units (*CU*) component, which may also be subject to failure, coordinates the *HARDWARE* based on requests received from the *SOFTWARE*. It has numerous interconnections both nominal and redundant with the wings, connections that are abstracted to 4 in Figure 9.1. It contains 4 power units (PCDU) and 4 thermal control units (TCU) that are responsible for the activation/deactivation of the TKs, each of them being connected to two different wings. Two command and monitoring units (CMU) supervise the entire system, i.e. all requests from the software transit the CMUs.

The SGS describes two operating modes: (1) the deployment of the solar arrays and (2) their rotation. We are interested here only in the first mode. Initially the four solar arrays of the ATV are stowed. Their deployment starts by removing the safety barriers from the thermal control units. Safety barriers prevent an unwanted unfolding of the wings by blocking the enabling of the thermal knives. Next the HDRSs are cut by at least 4 of the 8 thermal knives of each wing. In order for a HDRS to be cut, the knife has to be active for 50 consecutive time units. The deployment of the wing starts immediately after the last HDRS is cut. After the deployment is completed, the safety barriers are restored.

The system's redundancy is explicitly modeled for the TKs and HDRSs of each wing, TCUs and PCDUs, in case of anomaly at execution. There are 56 possible failures and each may occur at an arbitrary moment during the execution. The hypothesis is that the system may be subject to at most one failure, i.e. 1-fault tolerance. In order to ease the generation of verification configurations, a special *SIMULATION* component is added to the model to command non-deterministically the failure of an equipment based on a parameter that can be provided prior to the verification session.

The initial SGS model was realized using standard SysML modeling which does not impose strong static typing rules. Therefore, several modifications were made in order to make the model comply to the static well-formedness rules of OMEGA and correcting a range of inconsistencies which often go unnoticed throughout the system engineering phase and last sometimes until the integration test campaign, when correcting them becomes very costly. An example is provided by the lack of interface definition and bidirectional port modeling. This led us to model 18 interfaces for typing ports and correcting around 20% of the ports defined in the design that were not respecting either the unidirectionality rule or the uniqueness rule for outgoing connectors. The effort required for these syntactic changes is minor relative to the size of the project: between 1 and 2 person*days for the SGS model.

In terms of metrics, the obtained model defines a total of 21 block types (7 of which are refined by means of 24 Internal Block Diagrams) with 348 port types and 372 connector types for communication. At run-time, the system contains 96 block instances running in parallel with a total of 651 ports and 504 connectors.

We are interested in proving that the system is indeed 1-fault tolerant. Informally, it means that no matter which error occurs to equipment devices and at what moment, the software will attain the correct deployment of the wings. It is expressed by the following requirement modeled in Figure 9.2.

Requirement 9.1. *At the end of the deployment sequence, all four wings are deployed.*

We formalize this requirement with an observer. We add to the system model a block *Property* whose state machine describes the safety property to be verified: initially the observer waits in the state *SYSTEM_IS_ON* for the wing status

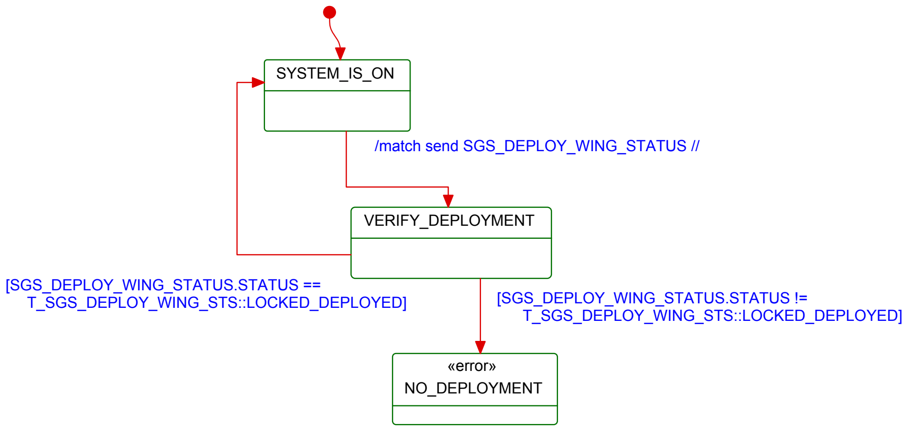

SYSTEM_IS_ON
/match send SGS_DEPLOY_WING_STATUS //
VERIFY_DEPLOYMENT
[SGS_DEPLOY_WING_STATUS.STATUS == T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED]
[SGS_DEPLOY_WING_STATUS.STATUS != T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED]
«error»
NO_DEPLOYMENT

Figure 9.2 – Observer formalizing Requirement 9.1: all four wings are deployed.

interrogation to be executed. After the entire sequence deployment is performed, a software piece verifies the locked status of the wings. When asked, the target wing answers with a *SGS_DEPLOY_WING_STATUS* message that has as parameter its current status. When the action is matched, the automaton passes into the *VERIFY_DEPLOYMENT* state where it checks the value of the parameter. If it is *LOCKED_DEPLOYED*, then it will wait for another occurrence of the interrogation for another or the same wing. Otherwise, something wrong has occurred at deployment and it advances to the error state *NO_DEPLOYMENT*. The reaching of the error state during verification means that the requirement is violated.

9.2 Preliminary Verification Results without Contracts[2]

We started by reviewing the system model and performing some preliminary validation and verification in order to detect modeling errors that may lead to the violation of the requirement. During this phase we did not yet make use of contracts, however we chose to review them here briefly, since they participated in eliminating some errors in the model and in showing that direct model-checking Requirement 9.1 is not feasible.

Interactive simulation of nominal scenarios and execution of random scenarios allowed us to discover several modeling errors. Most of them concerned unexpected message receptions that blocked the execution of components thus leading to general

[2] This section is based on [71].

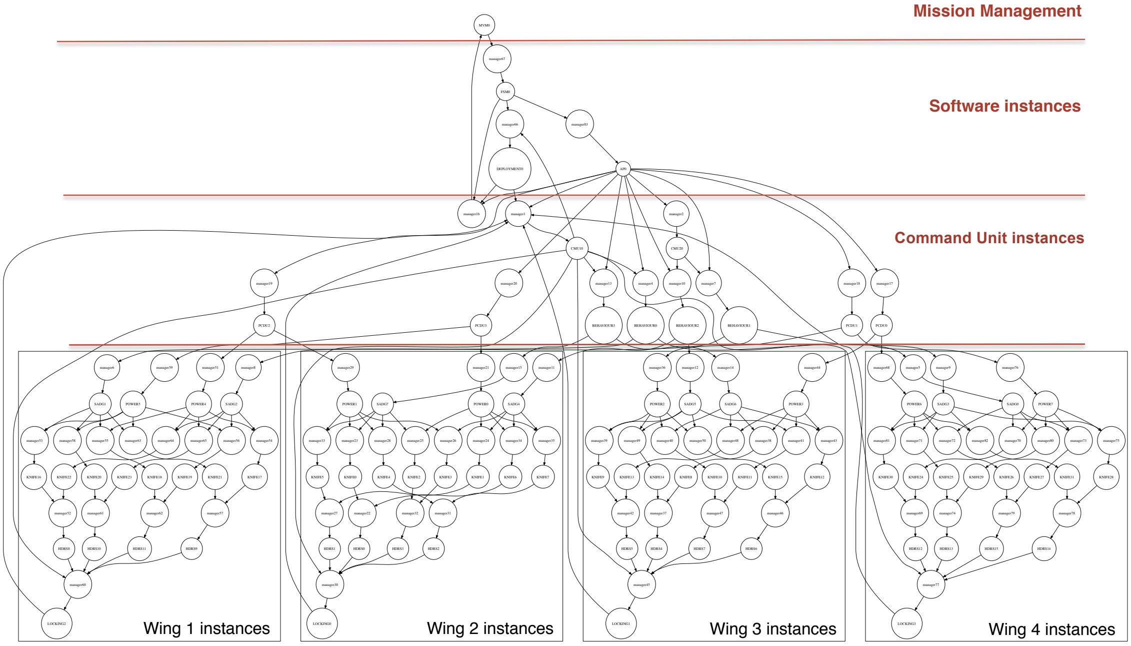


Figure 9.3 – System's communication graph displaying the components — represented as nodes — and their unidirectional communication — represented as arrows.

deadlocks, as it was the case for TKs. Message receptions had to be modeled for correct behavior. In other cases, like for the MVM, the unexpected message receptions were due to the existence of parallel composite states reacting to the same request: the designer intended only one parallel state to react to the message during a certain flight phase, but failed to correctly specify the conditions that enabled and disabled the receptions when going from one phase to another. We mention that an examined system scenario has around 2400 transitions fired and needs around a minute to be executed with the IFx simulator on a regular desktop machine.

We also inspected formally the system model for the absence of deadlocks since simulation allows only to partially explore the system's execution. This step permitted to discover missing timing constraints but for the rotation phase of the flight, which is not considered by the requirement to satisfy.

Performing model-checking on the current configuration is not possible[3] due to the combinatorial explosion of the state space. This is caused by the large number of component instances at run-time together with the large number of failures that need to be checked, as it can be seen in the communication graph represented

[3] The state space generation had to be stopped after 8 hours as the memory was exhausted.

in Figure 9.3. As a first way around the explosion problem, we used in the beginning a non-exhaustive exploration by limiting concurrency in the system to two threads, one for the *SIMULATION* component and one for all the other components. This allowed us to discover several missing transitions for TKs and, most importantly, incorrect connections between the PCDU and the wings. Each PCDU was erroneously connected to the same wing by both connections while it had to be nominally connected to one wing and redundantly connected to another wing.

Once corrections were made to the model, the exploration of the state space in the 2-thread configuration produced no further errors. However, this is not sufficient to establish the satisfaction of the requirement in the general case. For this reason we set out to use contract-based reasoning, which is described in the following section.

9.3 Applying the Contract-based Verification Technique

We start by identifying the components that represent the system under study S and the environment E. Since Requirement 9.1 is expressed with respect to the behavior of the four wings that are contained in the *HARDWARE* block, with regard to the methodology of Figure 3.1, we consider the subsystem S to be the *HARDWARE* and the K_i the *WINGi*, $i = \overline{1,4}$. The environment of the subsystem is given by the parts with which it communicates: bidirectional communication is established between *CU* and *HARDWARE*, while *CU* depends on the behavior of *SOFTWARE* and *MVM*. So, the environment E of Figure 3.1 is represented here by the composition of *MVM*, *SOFTWARE* and *CU*.

We model a set of contracts $\{\mathcal{C}\_W1, \dots, \mathcal{C}\_W4\}$, one for each wing, and a dominated contract $\mathcal{C}\_HW$ for the *HARDWARE* component. Each component is linked to its contract by a *contractUse* relation, as illustrated in Figure 9.4, where the *reqTarget* is set to *Property* and $\mathcal{C}\_HW$ as *refTarget*. Finally, the top contract is linked by a *contractConformance* to the *Property* it must satisfy.

The first step of the methodology consists in defining the contract $\mathcal{C}\_Wi = (A\_Wi, G\_Wi)$ for each *WINGi*, and next proving that *WINGi* satisfies $\mathcal{C}\_Wi$, $i = \overline{1,4}$. We chose for *WINGi* to use as assumption the concrete environment of the subsystem *HARDWARE* composed with an abstraction *WAj* for each *WINGj* with $j \neq i$. We propose the following abstraction *WAj*: the wing consumes all

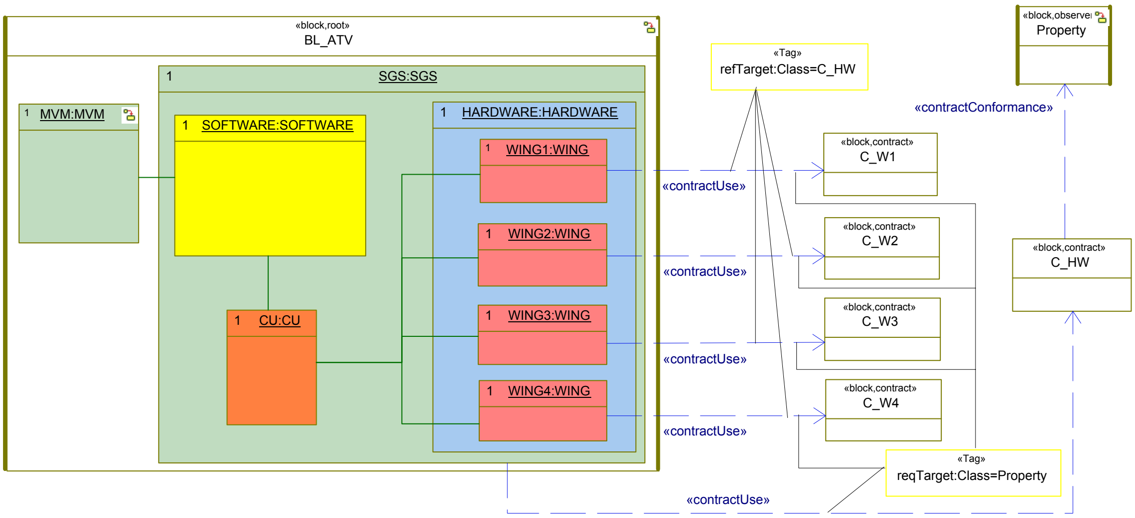


Figure 9.4 – The SGS model extended with contracts for verifying Requirement 9.1.

requests coming from the environment, and answers to any status request with *deployed*. Then the assumption A_Wi is given by the parallel composition of MVM, $SOFTWARE$, CU and WAj with $j \neq i$. This abstraction of the environment is sufficient to drastically reduce the state space of the verification model, since the exponential explosion in the original model is mainly due to the parallelism of the hardware pieces which are abstracted to the three leaf parts WAj. We can evaluate this reduction only with respect to the number of model elements instances — blocks, ports and connectors: the system instances are reduced in average by 55%. We want to guarantee that even if $WINGi$ exhibits a failure it ends up being deployed.

Contract $\mathcal{C}\_Wi = (A\_Wi, G\_Wi)$ where:

- $A\_Wi = MVM \parallel SOFTWARE \parallel CU \parallel (\parallel_{j \neq i} WAj)$.
- $G\_Wi = WAi$: the wing answers to requests about its status with *deployed* and ignores all other requests.

The contract is modeled in Figure 9.5, while Figure 9.6 presents the behavior of the guarantee. We note that since we use as assumption the concrete environment, the signature of the guarantee remains the same as that of the component. For this reason, we have to add consuming transitions in every state for all inputs corresponding to the wing deployment process. Moreover, we remark that the same guarantee type is used within all 4 modeled contracts; this shows the reusability of our contract modeling framework. Also, the guarantee G_Wi has the same

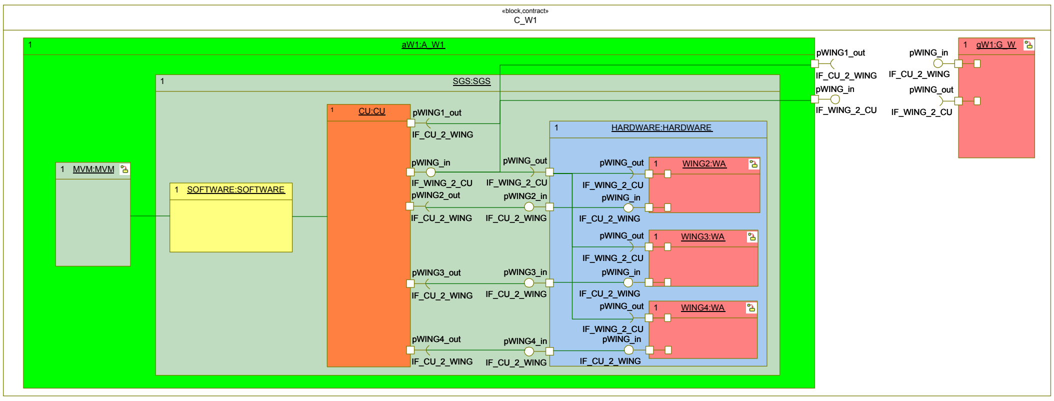


Figure 9.5 – The contract $\mathcal{C}\_W1$ for $WING1$ in SysML.

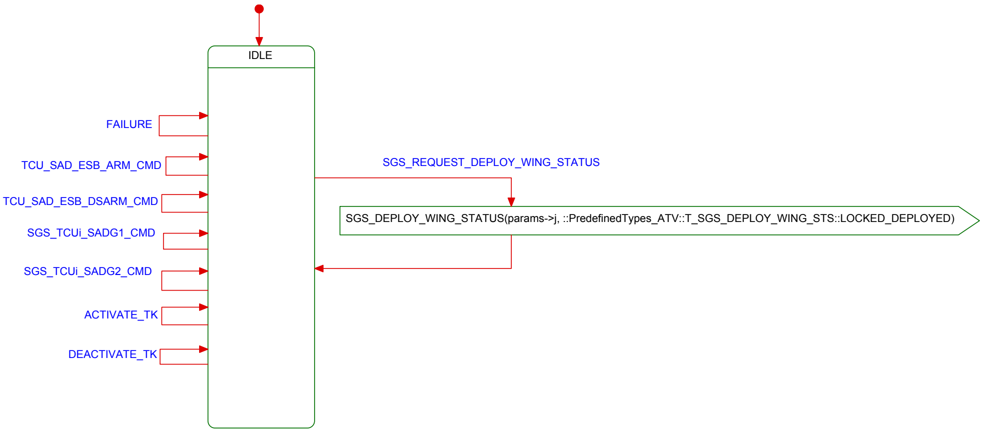

IDLE
FAILURE
TCU_SAD_ESB_ARM_CMD
TCU_SAD_ESB_DSARM_CMD
SGS_TCUi_SADG1_CMD
SGS_TCUi_SADG2_CMD
ACTIVATE_TK
DEACTIVATE_TK
SGS_REQUEST_DEPLOY_WING_STATUS
SGS_DEPLOY_WING_STATUS(params->j, ::PredefinedTypes_ATV::T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED)

Figure 9.6 – The modeled behavior for all G_Wi and G — parameter j ranges through 1 to 4.

behavior as the abstraction WAi. We prefer not to introduce a new notation which is already cumbersome and we will use WAi to denote also the guarantees. Furthermore, one can remark that the guarantee is stronger than the projection of the Requirement 9.1 on $WINGi$: the abstraction WAj can also be subject to one failure since this case was not excluded from its behavior; so, the fault tolerance property that we verify via contracts is stronger than the one intended. We guarantee that the system is 4-fault tolerant if faults occur in separate wings.

Next, we model the global contract $\mathcal{C}\_HW = (A\_HW, G\_HW)$ for $HARDWARE$ and we prove that the contract is dominated by $\{\mathcal{C}\_W1, \mathcal{C}\_W2, \mathcal{C}\_W3, \mathcal{C}\_W4\}$, i.e. the second step of the methodology. Again, we use as assumption A_HW the concrete environment of $HARDWARE$. The guarantee G_HW is the composition of the four WAi. In fact, G_HW has the same type as G_Wi since the same state machine can be used to describe its behavior.

Contract $\mathcal{C}\_HW = (A\_HW, G\_HW)$ where:

- $A\_HW = MVM \parallel SOFTWARE \parallel CU$
- G_HW : for each wing status interrogation answers with *deployed*, while all other requests are ignored.

All WAi, $i = \overline{1,4}$, and, in consequence, G_HW as defined satisfy the closure conditions for applying Theorem 6.5. In consequence, the following proof obligations need to be verified:

(2.1) $WA1 \parallel WA2 \parallel WA3 \parallel WA4 \sqsubseteq_{MVM \parallel SOFTWARE \parallel CU} G\_HW$

(2.2) $MVM \parallel SOFTWARE \parallel CU \parallel WA2 \parallel WA3 \parallel WA4 \sqsubseteq_{WA1} MVM \parallel SOFTWARE \parallel CU \parallel WA2 \parallel WA3 \parallel WA4$

(2.3) $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA3 \parallel WA4 \sqsubseteq_{WA2} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA3 \parallel WA4$

(2.4) $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA4 \sqsubseteq_{WA3} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA4$

(2.5) $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA3 \sqsubseteq_{WA4} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA3$

We remark that the last 4 items are trivial since we have the same member on both sides of the relations. Then only item (2.1) has to be automatically verified.

The third step of the reasoning consists in proving the satisfaction of the "mirror" contract $\mathcal{C}\_HW^{-1}$. This verification is trivial since the concrete environment is used as assumption and the proof obligation is written: $MVM \parallel SOFTWARE \parallel CU \sqsubseteq_{G\_HW} MVM \parallel SOFTWARE \parallel CU$.

The last step consists in verifying that $\mathcal{C}\_HW$ conforms to Requirement 9.1:
$MVM \parallel SOFTWARE \parallel CU \parallel G\_HW \preceq Property$.

The proofs of steps 1, 2 and 4 have been automatically performed within the IFx2 Toolset with the method described in Section 6.5. For each step of the verification methodology we have manually modeled the contracts: assumptions as composite blocks that we had to connect via ports with the other components and one type of guarantee that we have instantiated in every contract. Since our compiler does not yet generate the complete proof obligations, we had to manually provide the verification configurations for each step.

The implementation step resulted in 4 possible configurations with one concrete wing and 3 abstract ones that were each verified with respect to all 14 possible failures. The average time in seconds needed for the verification of the satisfaction

	Average verification time (s)			
Type of induced failure	Wing 1	Wing 2	Wing 3	Wing 4
Thermal knife	13993	6869	18842	11412
Hold-down and release system	12672	6516	16578	9980
Solar array driving group	11527	5432	13548	6807

Table 9.1 – Average verification time for each contract $\mathcal{C}\_Wi$ per induced failure group.

relation for each contract with respect to each class of failures is presented in Table 9.1. Even though the system model looks symmetrical, the command units do not have a symmetrical behavior and, due to their interconnections with the wings, the state space of system's abstraction for *WING1* and *WING3* is larger than the one of *WING2* and *WING4*: the *CMU1* is responsible for *WING1* and *WING3* during wing deployment but transfers requests to the four wings during preparation, whereas *CMU2* handles only the wing deployment for *WING2* and *WING4*.

It is interesting to note that these results have been obtained after the model was corrected: the first verification of contract satisfaction failed with a counterexample when injecting a failure on TKs due to their parallel execution. Deployment starts by switching on a subset of nominal TKs for each wing. The TK will inform the HDRS of its enabling. If, in the 50 seconds needed for the cut, the redunant TK that is off fails, it will send the disabling command to the HDRS, and the HDRS won't be cut. Now, the redundant TK cannot be enabled due to the failure mode, the cable is not cut and the wing not deployed. The correction consisted in sending the switch off request in case of a failure only if the TK is on.

For the dominance step, the only proof obligation to be verified is the refinement of the global guarantee — item (2.1) —, which took 1 second.

Finally, on the only model configuration provided by dominance, we proved the conformance of $\mathcal{C}\_HW$ to Requirement 9.1 that also took 1 second.

9.4 Conclusion

This case study, extracted from an industrial-scale system, provides the practical motivation for our work. It clearly shows the limitations of current verification techniques for proving the satisfaction of system requirements which is a compelling burden for the correct development of systems with respect to their safety, reliability, availability, etc.

In the effort of tackling these issues, we have applied the contract-based framework we developed on the case study. So, we have defined a set of contracts and we automatically verified each proof obligation. Once the system was apprehended, the definition of contracts was rather straightforward: the global guarantee is almost identical to the requirement to prove, while the component guarantees are the projection of the latter on each component. From our view, the expected overhead in this case study was moderate: it is directly related to the effort of understanding the developed system and of modifying the system model to comply to the strong static typing rules imposed by the component framework. Moreover, this approach has allowed us to detect some intricate modeling errors and, finally, to conclude that the system satisfies indeed its requirement.

Based on the positive feedback with respect to the verification results, we can ascertain that previously intractable models can be tamed by the contract-based verification methodology and technique described in this thesis.

Conclusion and Perspectives

In order to tackle the growing complexity of nowadays critical real-time systems, we considered in this thesis a compositional component-based design methodology driven by requirements. In order to obtain directly from requirements a correct system design, we examined an intermediate layer in the form of a contract framework that allows to specify, in an abstract manner, how a requirement can be decomposed on components and how components contribute to the satisfaction of a requirement. Therefore, the notion of contract consists in a pair (assumption, guarantee) and it is involved in three refinement relations that allow for iterative design, namely: conformance verifies that the contract refines the global requirement, dominance verifies that a general contract is refined by a set of specific ones and satisfaction verifies that the implementation satisfies its contract. These generic notions are structured in a contract-based reasoning methodology in [143, 144, 142] that is instantiated throughout this thesis.

We developed a behavioral contract framework for system designs modeled with SysML, which can also be used for the compositional verification of timed safety requirement satisfaction. Our contribution is two-fold: first, we have introduced the generic contract framework by syntax in SysML and secondly, we have defined its semantics based on a variant of Timed Input/Output Automata and we sketched a verification method based on model-checking in order prove that the modeled refinement relations are satisfied.

Overall, the presented contract-based theory aims to offer the following features:

- scalability: the methodology can be applied on large real-life systems and produce a yes/no satisfaction answer for the global requirements,
- predictability: design errors may be detected from the early phases of the system design, while the method can be applied from coarse-grained to fine-grained architectural models, and
- reusability both for the design, e.g. contracts are defined by instances of type which are at their turn reusable, and the verification — a component/contract

can be replaced with another one as long as refinement locally holds.

Moreover, it supports an incremental and independent development of system designs.

Contracts in SysML

In the first place, since SysML is a rich modeling standard, we have selected a subset of its modeling elements sufficient to describe hierarchical component-based systems. Moreover, being a semi-formal modeling language which leaves open several semantic variation points that result in an ambiguous model semantics, we have presented a component framework based on SysML, called OMEGA, which allows for rigorous system engineering by requiring for a set of strong typing well-formedness rules to hold. This component language is extended with a continuous time base and clock notions and a mechanism for formalizing and verifying timed safety system requirements with observers.

Next, we have introduced the contract-related notions by defining a domain meta-model enriched with well-formedness rules such that a system design extended with contracts is unambiguous and strongly typed. The meta-model is described using the UML standard and, therefore, the proposed extension can be generally applied on any UML/SysML model. The syntactic definition of the contract-based framework is generic enough to be used with other defined semantical frameworks, provided that they can represent the semantics of a SysML model. Our definition of contracts explicitly handles an important aspect of the requirement-driven design: components/contracts can have a larger signature than their corresponding abstract version, which is due to either incorporating several requirements into an implementation or detailing the component's contribution toward the satisfaction of the requirement.

We have instantiated the meta-model for the OMEGA component language by using the stereotype mechanism in order to make the extension usable with standard model editors and we have formalized the meta-model's well-formedness rules using OCL such that system designs extended with contracts can be statically checked for consistency and coherence. The key concepts and design methodology have been illustrated of a simplified version of an Automated Teller Machine (sATM).

In order to keep the description simple, we assumed some restrictions on the component model which do not have an impact on the expressiveness of the design, like all communications bypass ports and one port can be typed with only one interface. Future work should explore the relaxation of these conditions. For example, an idea which is available in SysML is to allow ports to express different functionalities with respect to different components by typing them with several interfaces, possibly hierarchically structured for a clear typing system. Then, the contract signature refinement would be transferred from port definition to port type definition. Indeed, it would be interesting to allow the user to redefine the type of a port from the guarantee by considering only a subset. Yet, such modeling will also transfer the design complexity from ports to types and may induce an important overhead at compilation for contract satisfaction during dominance, since the target for signals has to be automatically computed based on ports and connectors and for which type verification would be necessary.

Formal Contract-based Theory with Timed Input/Output Automata

The novelty for this notion of contract is given by the behavioral aspect of the assumption/guarantee which expresses properties of the whole dynamics of a component via the state machine they can model. Therefore, verifying the behavioral refinement relations a contract is involved in requires to formalize the semantics of the component language. The second goal of our work consisted in providing a suitable semantic framework in the form of Timed Input/Output Automata for the component language extended with contracts. We have defined a variant of the TIOA framework of [108] such that it complies to the semantics of the component model and we have established as refinement relation between components the timed trace inclusion, which is preserved by composition. Subsequently, we have presented the mapping of SysML modeling elements to TIOA and we have sketched an algorithm for the generation of proof obligations.

Upon the TIOA component framework, we have built the contract-based framework by defining the semantics of proof obligations, i.e. refinement under context on which both contract satisfaction and dominance are based. At its turn, refinement under context relies on the conformance relation, in our case timed trace inclusion. The reasoning with this contract framework is both well-defined and sound for timed safety properties, fact which is established by the compositionality results the theory satisfies, i.e. refinement under context is preserved by composition and guarantees the correctness of circular reasoning.

Since the associated proof obligations consist in verifying a set of timed trace inclusion relations and trace inclusion is undecidable (except for some categories of TA), we have presented a verification method for a sub-class of safety properties that allows to check trace inclusion by model-checking. So, a guarantee is transformed into a timed property automaton (an observer) and we show that this transformation is sufficient in order to prove trace inclusion. Yet, the verification method is limited to deterministic safety properties. A particular attention must be given to contract assumptions, which, different from guarantees, are not required to be modeled as safety properties by the formal framework. Recall that assumptions play the role of a timed safety property when proving dominance. In consequence, modeling assumptions as timed safety properties may turn out difficult if the environment's behavior they abstract exhibits hard timed upper bounds on actions.

Future work concerns the enlarging or strengthening of the type of requirements that can be verified by the contract-based approach. The motivation is given by the language that can be used to express safety properties, which does not allow to specify hard timed bounds for actions. A solution would be to use timed simulation as conformance and/or refinement under context, relation which allows to express stronger times delays and possibly remove the closure under time-extension condition: the abstract component observes at least the same time elapse as the concrete component and, if it is explicitly modeled, then the lazyness of actions may be avoided. Also a simulation relation would permit to extend the type of requirements as for example with progress ones. An example of refinement under context relation based on simulation which verifies both safety and progress properties and ensures sound circular reasoning is given in [99, 98] for the BIP framework. Yet, two questions are raised by the usage of simulation: (1) how the refinement of signature can be taken into account and which semantics to use for actions that are not explicitly modeled in the guarantee and (2) how can it be automatically verified. With respect to the first item, we considered a covariant refinement of signature (i.e. more inputs and outputs in the component), where actions not appearing in the guarantee can be ignored. Yet, other interpretations are available: a co- and contravariant signature refinement (i.e. more inputs, less outputs in the component) may be used, while actions not explicitly modeled in the guarantee can be handled as errors. With respect to the second item, incorporating several types of requirements into a simulation relation would require to customize it and, so, induce an increased complexity for the automatic verification that is not equipped with a customizable tooling.

Implementation and Feedback on Experimental Results

The contract-based approach is partially implemented in the IFx2 toolset. The *uml2if* compiler automatically transforms an input system model with contracts into a network of TIOA, which can be model-checked and simulated within the toolset. The modeled refinement relations, as well as contracts, are not yet taken into consideration for generating the corresponding proof obligations, these steps remaining manual. For our experimentation, we have manually modeled different variations of the system that correspond to the left-hand side members of the obtained conformance relations, as well as the transformation from an assumption/guarantee defined as component into a safety property formalization. Future work consists in automating all described intermediate model generation steps and adding the functionality to manage the proof obligations.

Finally, we have illustrated our method on two case studies — a parametric one and another extracted from an industrial-scale system model — and we have shown how our approach can alleviate the problem of combinatorial explosion for the verification of large systems. This statement is supported by the positive feedback of the verification results.

We conclude that our contract-based framework is appropriate for the design and compositional verification of large critical real-time systems described by multiple hierarchical layers, which have to satisfy timed safety properties that do not model strong timed bounds for actions.

Automatic Generation of Contracts

An important issue which was not discussed in this thesis and which constitutes future work deals with the methodological guideline for modeling contracts. The lack of a well-defined technique which prescribes how to derive contracts for the whole system and the components is one of the reasons why previous attempts to introduce contracts in software engineering, merely programming languages, have not enjoyed an extensive popularity. Nevertheless, we believe that the case for contracts in the early phases of system engineering is different and that the contract concept is strongly needed for decomposing systems, as well as verifying their correctness before system implementation. We believe that applying verification techniques, even if the overhead for these applications is rather important, may be in most cases less expensive than finding errors in the system after implementation and deployment. Therefore, providing methods or methodological guidelines for deriving intermediate contracts from the properties one is trying to prove is our main perspective in the long run.

An idea would be to automatically generate the guarantees to use, based on (local) requirements. Indeed starting from a global requirement we could derive a first contract which incorporates in its signature specific (visible) actions that can be further used for contract decomposition. In general, this contract would need to be specified by the designer. Then from the guarantee we could project the global guarantee onto a set of sub-guarantees by taking into account the global assumption and having predefined a signature for each sub-component. The correctness of local guarantees with respect to the abstract one can be ensured with the CEGAR approach. We assume that such a step can be iteratively applied until the level of granularity corresponds to an implementation for that contract. Such an approach has been recently studied in [96] for LTS under the term of *realizability*.

This reasoning corresponds to the technique we have applied for finding the contracts throughout the cases studies of this thesis. In each example we started by identifying which actions the component performs that directly contribute to the requirement satisfaction, that also include the environment-dependent ones. The modeled guarantees are the weakest components which merely simulate the structure of their corresponding implementations. For the real-life system, the task was simple and straightforward: the global guarantee is almost identical to the requirement, while component guarantees represent a projection of the global one on components. Yet, this task requires to speculate an implementable component-based architecture and a set of possible actions.

In the same line of thought, we are interested in automatically generating the assumptions over the environment that are to be modeled for each contract. The reasoning is similar to generating the weakest pre-condition for a Hoare triplet: the guarantee G_i has been previously computed, either automatically or by definition, and the implementation K_i is provided; then an environment A_i such that $K_i \sqsubseteq_{A_i} G_i$ holds may be determined. If A_i cannot be computed it means that K_i is not a correct implementation for G_i. We could also make use of the proof obligations the dominance relation generates, if the implementation K_i is not provided: G, G_i and A are known; then we have to compute the weakest (and simplest) A_i such that the conformance relation roughly written $A \parallel (\parallel_{j=1}^{n} G_j) \preceq A_i \parallel G_i$ holds. How to compute A_i based on contract satisfaction has been studied in [84, 134, 83] based on CEGAR and automatic learning for LTS.

Such results would provide a fully automated technique for correct-by-construction system designs which could be integrated in an iterative component-based development process.

Error Diagnosis with Contracts

In the short run, a second perspective consists in exploring the error diagnosis topic for the contract-based framework, but also in its general. Indeed, shifting the obtained counterexample from the formal model to the semi-formal one will help system engineers to easily find the error and correct it. Bridging the two worlds would provide an asset for modeling abstractions — in the form of assumptions/guarantees in our case — and correcting them, whenever refinement does not locally hold.

Secondly, diagnosis should be performed on the entire proof tree, not only locally. We have sketched in Section 7.3 a set of indications of what to be done if one proof fails, which we have applied on the presented case studies for defining the correct abstractions. Yet, these rules should be thoroughly studied and ultimately defined as a methodology for diagnosis, which could be integrated in a development process.

Bibliography

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits W. Vaandrager. Automata Learning through Counterexample Guided Abstraction Refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, 2012.

[2] Martín Abadi and Gordon D. Plotkin. A Logical View of Composition and Refinement. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 323–332. ACM Press, 1991.

[3] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. Seeing Errors: Model Driven Simulation Trace Visualization. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 480–496. Springer, 2012.

[4] Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer Berlin Heidelberg, 1998.

[5] Luay Alawneh, Mourad Debbabi, Yosr Jarraya, Andrei Soeanu, and Fawzi Hassaïne. A Unified Approach for Verification and Validation of Systems and Software Engineering Models. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006), 27-30 March 2006, Potsdam, Germany*, pages 409–418. IEEE Computer Society, 2006.

[6] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 414–425. IEEE Computer Society, 1990.

[7] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[8] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[9] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating Refinement Relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[10] Charles André, Frédéric Mallet, and Robert de Simone. Modeling Time(s). In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2007.

[11] Pascal André, Gilles Ardourel, and Mohamed Messabihi. Vérification de contrats logiciels a l'aide de transformations de modeles. In *7èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, 2011.

[12] Ludovic Apvrille, Pierre de Saqui-Sannes, and Ferhat Khendek. TURTLE-P: a UML profile for the formal validation of critical and distributed systems. *Software and System Modeling*, 5(4):449–466, 2006.

[13] Alexandre Arnold, Benoît Boyer, and Axel Legay. Contracts and Behavioral Patterns for SoS: The EU IP DANSE approach. In Kim G. Larsen, Axel Legay, and Ulrik Nyman, editors, *Proceedings 1st Workshop on Advances in Systems of Systems, AiSoS 2013, Rome, Italy, 16th March 2013.*, volume 133 of *EPTCS*, pages 47–66, 2013.

[14] Roberto Barbuti and Luca Tesei. Timed automata with urgent transitions. *Acta Inf.*, 40(5):317–347, 2004.

[15] Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. Towards the UML-Based Formal Verification of Timed Systems. In Bernhard K.

Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2010.

[16] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 3–12. IEEE Computer Society, 2006.

[17] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from Specifications to Contracts in Component-Based Design. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.

[18] Sebastian S. Bauer, Rolf Hennicker, and Axel Legay. Component Interfaces with Contracts on Ports. In Pasareanu and Salaün [135], pages 19–35.

[19] Andreas Baumgart, Philipp Reinkemeier, Achim Rettberg, Ingo Stierand, Eike Thaden, and Raphael Weber. A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2010.

[20] Danièle Beauquier. On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003.

[21] Matthew Bennion and Ibrahim Habli. A candid industrial evaluation of formal software verification using model checking. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 175–184. ACM, 2014.

[22] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007.

[23] Albert Benveniste, Benoît Caillaud, and Roberto Passerone. A Generic Model of Contracts for Embedded Systems. *CoRR*, abs/0706.1456, 2007.

[24] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A Contract-based Formalism for the Specification of Heterogeneous Systems (invited). In *Forum on specification and Design Languages, FDL 2008, September 23-25, 2008, Stuttgart, Germany, Proceedings*, pages 142–147. IEEE, 2008.

[25] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, François Vernadat, et al. Fiacre: an intermediate language for model verification in the TOPCASED environment. In *ERTS 2008*, 2008.

[26] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. A Compositional Approach on Modal Specifications for Timed Systems. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 679–697. Springer, 2009.

[27] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. Modal event-clock specifications for timed component-based design. *Sci. Comput. Program.*, 77(12):1212–1234, 2012.

[28] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[29] Per Bjesse and James H. Kukula. Using Counter Example Guided Abstraction Refinement to Find Complex Bugs. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 156–161. IEEE Computer Society, 2004.

[30] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2008.

[31] Sébastien Bornot and Joseph Sifakis. An Algebraic Framework for Urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[32] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling Urgency in Timed Systems. In de Roever et al. [68], pages 103–129.

[33] Timothy Bourke, Alexandre David, Kim G. Larsen, Axel Legay, Didier Lime, Ulrik Nyman, and Andrzej Wasowski. New Results on Timed Specifications. In Till Mossakowski and Hans-Jörg Kreowski, editors, *Recent Trends in Algebraic Development Techniques - 20th International Workshop, WADT 2010, Etelsen, Germany, July 1-4, 2010, Revised Selected Papers*, volume 7137 of *Lecture Notes in Computer Science*, pages 175–192. Springer, 2010.

[34] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.

[35] Eric Cariou, Cyril Ballagny, Alexandre Feugas, and Franck Barbier. Contracts for Model Execution Verification. In France et al. [77], pages 3–18.

[36] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL contracts for the verification of model transformations. *ECEASST*, 24, 2009.

[37] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.

[38] Karlis Cerans, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed Modal Specification - Theory and Tools. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1993.

[39] Taolue Chen, Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. A Compositional Specification Theory for Component Behaviours. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2012.

[40] Zhe Chen, Yi Gu, Zhiqiu Huang, Jun Zheng, Chang Liu, and Ziyi Liu. Model checking aircraft controller software: a case study. *Software: Practice and Experience*, 2013.

[41] Shing-Chi Cheung and Jeff Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[42] Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. Assume-Guarantee Reasoning for Safe Component Behaviours. In Pasareanu and Salaün [135], pages 92–109.

[43] Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. Compositional assume-guarantee reasoning for input/output component theories. *Sci. Comput. Program.*, 91:115–137, 2014.

[44] Chris Chilton, Marta Z. Kwiatkowska, and Xu Wang. Revisiting Timed Specification Theories: A Linear-Time Perspective. In Marcin Jurdzinski and Dejan Nickovic, editors, *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*, volume 7595 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2012.

[45] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen,*

Denmark, July 27-31, 2002, Proceedings, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[46] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 702–705. IEEE, 2013.

[47] Alessandro Cimatti and Stefano Tonetta. A Property-Based Proof System for Contract-Based Design. In Vittorio Cortellessa, Henry Muccini, and Onur Demirörs, editors, *38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 21–28. IEEE Computer Society, 2012.

[48] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, (0):–, 2014.

[49] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[50] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[51] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[52] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[53] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional Model Checking. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science, 5-8 June, 1989, Asilomar Conference Center, Pacific Grove, California, USA*, pages 353–362. IEEE Computer Society, 1989.

[54] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning Assumptions for Compositional Verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[55] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional Verification of Architectural Models. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2012.

[56] Benoît Combemale, Laure Gonnord, and Vlad Rusu. A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In France et al. [77], pages 35–51.

[57] Éric Conquet, François-Xavier Dormoy, Iulia Dragomir, Susanne Graf, David Lesens, Piotr Nienaltowski, and Iulian Ober. Formal Model Driven Engineering for Space Onboard Software. *Embedded Real Time Software and Systems (ERTS2)*, 2012.

[58] Éric Conquet, François-Xavier Dormoy, Iulia Dragomir, Alain Le Guennec, David Lesens, Piotr Nienaltowski, and Iulian Ober. Modèles système, modèles logiciel et modèles de code dans les applications spatiales. *Génie logiciel*, 97:9–15, 2011.

[59] Eduardo Correia da Silva and Emilia Villani. Integrando SysML e model checking para V&V de software crítico espacial. In *Brasilian Symposium on Aeropspace Engineering and Applications, São José dos Campos, SP, Brazil*, 2009.

[60] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 1023–1028. IEEE, 2011.

[61] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Methodologies for Specification of Real-Time Systems Using

Timed I/O Automata. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *FMCO*, volume 6286 of *Lecture Notes in Computer Science*, pages 290–310. Springer, 2009.

[62] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In Karl Henrik Johansson and Wang Yi, editors, *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 91–100. ACM, 2010.

[63] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Mikael H. Møller, Ulrik Nyman, Anders P. Ravn, Arne Skou, and Andrzej Wasowski. Compositional verification of real-time systems using ECDAR. *STTT*, 14(6):703–720, 2012.

[64] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2010.

[65] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 109–120. ACM, 2001.

[66] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.

[67] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed Interfaces. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[68] Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97,*

Bad Malente, Germany, September 8-12, 1997. Revised Lectures, volume 1536 of *Lecture Notes in Computer Science*. Springer, 1998.

[69] Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. Automatic Abstraction Refinement for Timed Automata. In Raskin and Thiagarajan [146], pages 114–129.

[70] Iulia Dragomir and Iulian Ober. Well-formedness and typing rules for UML Composite Structures. *CoRR*, abs/1010.6155, 2010.

[71] Iulia Dragomir, Iulian Ober, and David Lesens. A Case Study in Formal System Engineering with SysML. In Isabelle Perseil, Karin Breitman, and Marc Pouzet, editors, *17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20, 2012*, pages 189–198. IEEE Computer Society, 2012.

[72] Iulia Dragomir, Iulian Ober, and Christian Percebois. Integrating verifiable Assume/Guarantee contracts in UML/SysML. In Iulian Ober, Florian Noyrit, Susanne Graf, and Gabor Karsai, editors, *ACESMB@MoDELS*, volume 1084 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[73] Iulia Dragomir, Iulian Ober, and Christian Percebois. Integrating Verifiable Assume/Guarantee Contracts in UML/SysML. Technical Report IRIT/RT-2013-14-FR, IRIT, July 2013.

[74] Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety contracts for timed reactive components (extended abstract). In Laurence Duchien, editor, *Journées nationales du GDR CNRS Programmation et Logiciel, Nancy, 03/04/2013-05/04/2013*, pages 37–46. Université de Loraine, avril 2013.

[75] Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety contracts for timed reactive systems. Technical Report IRIT/RT-2013-11-FR, IRIT, June 2013.

[76] Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety Contracts for Timed Reactive Components in SysML. In Viliam Geffert, Bart Preneel, Branislav Rovan, Julius Stuller, and A Min Tjoa, editors, *SOFSEM*, volume 8327 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2014.

[77] Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors. *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, volume 6698 of *Lecture Notes in Computer Science*. Springer, 2011.

[78] Andrew Gacek, Andreas Katis, Michael W. Whalen, and Darren Cofer. Hierarchical Circular Compositional Reasoning. Technical Report 2014-1, University of Minnesota Software Engineering Center, 200 Union St., Minneapolis, MN 55455, March 2014.

[79] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[80] Ning Ge and Marc Pantel. Time Properties Verification Framework for UML-MARTE Safety Critical Real-Time Systems. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2012.

[81] Marc Geilen, Stavros Tripakis, and Maarten Wiggers. The earlier the better: a theory of timed actor interfaces. In Marco Caccamo, Emilio Frazzoli, and Radu Grosu, editors, *Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, IL, USA, April 12-14, 2011*, pages 23–32. ACM, 2011.

[82] Abdelouahed Gherbi and Ferhat Khendek. UML Profiles for Real-Time Systems and their Applications. *Journal of Object Technology*, 5(4):149–169, 2006.

[83] Dimitra Giannakopoulou and Corina S. Pasareanu. Abstraction and Learning for Infinite-State Compositional Verification. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Festschrift for Dave Schmidt*, volume 129 of *EPTCS*, pages 211–228, 2013.

[84] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption Generation for Software Component Verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 3–12. IEEE Computer Society, 2002.

[85] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Component Verification with Automatically Generated Assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.

[86] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[87] Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *STTT*, 8(2):113–127, 2006.

[88] Susanne Graf and Sophie Quinton. Contracts for BIP: Hierarchical Interaction Models for Compositional Verification. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[89] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification (SPT) v1.0, 2003.

[90] Object Management Group. Systems Modelling Language (SysML) v1.1, 2008.

[91] Object Management Group. Unified Modeling Language (UML) v2.2, 2009.

[92] Object Management Group. Object Constraint Language (OCL) v2.2, 2010.

[93] Object Management Group. XML Metadata Interchange (XMI) v2.4, 2014.

[94] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[95] Orna Grumberg and David E. Long. Model Checking and Modular Verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[96] Matthias Güdemann, Gwen Salaün, and Meriem Ouederni. Counterexample Guided Synthesis of Monitors for Realizability Enforcement. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2012.

[97] Anubhav Gupta and Edmund M. Clarke. Reconsidering CEGAR: Learning Good Abstractions without Refinement. In *23rd International Conference*

on Computer Design (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, pages 591–598. IEEE Computer Society, 2005.

[98] Imene Ben Hafaiedh. *Systèmes à base de composants : du design à l'implémentation.* PhD thesis, Université de Grenoble, 2011.

[99] Imene Ben Hafaiedh, Susanne Graf, and Sophie Quinton. Reasoning about Safety and Progress Using Contracts. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2010.

[100] Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996.

[101] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed Transition Systems. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz 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*, pages 226–251. Springer, 1991.

[102] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal Proof Methodologies for Timed Transition Systems. *Inf. Comput.*, 112(2):273–337, 1994.

[103] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Inf. Comput.*, 111(2):193–244, 1994.

[104] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[105] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[106] International Telecommunication Union (ITU). Specification and Description Language (SDL-2010). ITU-T Recommandation Z.100, December 2011.

[107] International Telecommunication Union (ITU). Specification and Description Language - Unified modeling language profile for SDL-2010. ITU-T Recommandation Z.109, October 2013.

[108] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata, Second Edition.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[109] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking - Timed UML State Machines and Collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.

[110] Daniel Knorreck, Ludovic Apvrille, and Pierre de Saqui-Sannes. TEPE: a SysML language for time-constrained property modeling and formal verification. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[111] Alexander Kraas. The SDL-UML Profile Revisited. In Frank Alexander Kraemer and Peter Herrmann, editors, *System Analysis and Modeling: About Models*, volume 6598 of *Lecture Notes in Computer Science*, pages 108–123. Springer Berlin Heidelberg, 2011.

[112] Alexander Kraas and Patrick Rehm. Results in using the new version of the SDL-UML profile. In *Joint ITU-T and SDL Forum Society Workshop on ITU System Design Languages, Geneva, Switzerland*, September 2008.

[113] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[114] Matthias P. Krieger, Alexander Knapp, and Burkhart Wolff. Automatic and efficient simulation of operation contracts. In Eelco Visser and Jaakko Järvi, editors, *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, pages 53–62. ACM, 2010.

[115] Barath Kumar and Jürgen Jasperneite. UML Profiles for Modeling Real-Time Communication Protocols. *Journal of Object Technology*, 9(2):178–198, 2010.

[116] Kim Guldstrand Larsen. Modal Specifications. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[117] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface Input/Output Automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[118] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On Modal Refinement and Consistency. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2007.

[119] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.

[120] Mohamed Messabihi, Pascal André, and Christian Attiogbé. Multilevel Contracts for Trusted Components. In Javier Cámara, Carlos Canal, and Gwen Salaün, editors, *Proceedings International Workshop on Component and Service Interoperability, WCSI 2010, Málaga, Spain, 29th June 2010.*, volume 37 of *EPTCS*, pages 71–85, 2010.

[121] Bertrand Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25(10):40–51, 1992.

[122] Erich Mikk, Yassine Lakhnechi, and Michael Siegel. Hierarchical automata as model for statecharts. In R.K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science - ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer Berlin Heidelberg, 1997.

[123] Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989.

[124] Ileana Ober, Iulian Ober, Iulia Dragomir, and El Arbi Aboussoror. UML/SysML semantic tunings. *ISSE*, 7(4):257–264, 2011.

[125] Iulian Ober and Iulia Dragomir. OMEGA2: A New Version of the Profile and the Tools. In Radu Calinescu, Richard F. Paige, and Marta Z. Kwiatkowska, editors, *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010*, pages 373–378. IEEE Computer Society, 2010.

[126] Iulian Ober and Iulia Dragomir. Unambiguous UML Composite Structures: The OMEGA2 Experience. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolic, and Stefan Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, volume 6543 of *Lecture Notes in Computer Science*, pages 418–430. Springer, 2011.

[127] Iulian Ober, Susanne Graf, and David Lesens. Modeling and Validation of a Software Architecture for the Ariane-5 Launcher. In Roberto Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, volume 4037 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2006.

[128] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *STTT*, 8(2):128–145, 2006.

[129] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems v1.1, 2011.

[130] Object Management Group. Semantics of A Foundational Subset For Executable UML Models (FUML) v1.1, 2013.

[131] Joël Ouaknine and James Worrell. On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 54–63. IEEE Computer Society, 2004.

[132] David Lorge Parnas and David M. Weiss. Active design reviews: Principles and practices. *Journal of Systems and Software*, 7(4):259–265, 1987.

[133] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 1999.

[134] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and

conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[135] Corina S. Pasareanu and Gwen Salaün, editors. *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, volume 7684 of *Lecture Notes in Computer Science*. Springer, 2013.

[136] Richard Payne and John Fitzgerald. Contract-based interface specification language for functional and non-functional properties. Technical report, Newcastle University, 2011.

[137] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. AVATAR: A SysML environment for the formal verification of safety and security properties. In *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*, pages 1–10. IEEE, 2011.

[138] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[139] Louchka Popova-Zeugmann. On time petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4):227–244, 1991.

[140] Imran Rafiq Quadri, Etienne Brosse, Ian Gray, Nikolas Drivalos Matragkas, Leandro Soares Indrusiak, Matteo Rossi, Alessandra Bagnato, and Andrey Sadovykh. MADES FP7 EU project: Effective high level SysML/MARTE methodology for real-time and embedded avionics systems. In Leandro Soares Indrusiak, Guy Gogniat, and Nikolaos S. Voros, editors, *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), York, United Kingdom, July 9-11, 2012*, pages 1–8. IEEE, 2012.

[141] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[142] Sophie Quinton. *Design, vérification est implémentation de système à composants.* PhD thesis, Université de Grenoble, 2011.

[143] Sophie Quinton and Susanne Graf. A framework for contract-based reasoning: Motivation and application. In *FLACOS'08 Second Workshop on Formal*

Languages and Analysis of Contract-Oriented Software, volume 7, page 77, 2008.

[144] Sophie Quinton and Susanne Graf. Contract-Based Verification of Hierarchical Systems of Components. In Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 377–381. IEEE Computer Society, 2008.

[145] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces: unifying interface automata and modal specifications. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 87–96. ACM, 2009.

[146] Jean-François Raskin and P. S. Thiagarajan, editors. *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, volume 4763 of *Lecture Notes in Computer Science*. Springer, 2007.

[147] Samuel Rochet and Yves Bernand. AGATE, a transfomation tool for simulation and formal checking for UML/SysML activity (Abstract). In *TOPCASED Days, Toulouse*, february 2011.

[148] SAE. Architecture Analysis and Design Language (AADL), 2004. Document No. AS5506/1.

[149] SDL-RT. Specification and Description Language - Real Time (SDL-RT), v2.3. ITU-T Recommandation Z.100, April 2013.

[150] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada, June 1998, Proceedings*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.

[151] SPEEDS. D 2.5.4: Contract Specification Language, 2008.

[152] Jagadish Suryadevara, Cristina Cerschi Seceleanu, Frédéric Mallet, and Paul Pettersson. Verifying MARTE/CCSL Mode Behaviors Using UPPAAL. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods - 11th International Conference,*

SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings, volume 8137 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[153] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A Theory of Synchronous Relational Interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14, 2011.

[154] Farn Wang. Symbolic Simulation-Checking of Dense-Time Automata. In Raskin and Thiagarajan [146], pages 352–368.

[155] Ting Wang, Jun Sun, Yang Liu, Xinyu Wang, and Shanping Li. Are Timed Automata Bad for a Specification Language? Language Inclusion Checking for Timed Automata. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2014.

[156] Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau. A UML Meta-model for Contract Aware Components. In Martin Gogolla and Cris Kobryn, editors, *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 442–456. Springer, 2001.

[157] Michael W. Whalen, Andrew Gacek, Darren D. Cofer, Anitha Murugesan, Mats Per Erik Heimdahl, and Sanjai Rayadurgam. Your 'What' Is My 'How': Iteration and Hierarchy in System Design. *IEEE Software*, 30(2):54–60, 2013.

Appendices

A OCL Formalization of the Well-Formedness Set of Rules for Contracts in UML/SysML

A.1 Rules Defined on the Meta-Model of Contracts

```
MainModel : http://www.eclipse.org/uml2/3.0.0/UML

context Namespace

def: getDependenciesRec : Set(Dependency) =
     self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{}
        | if m.oclIsTypeOf(uml::Dependency) then res->union(m.
       oclAsType(uml::Dependency)->asSet())
          else if m.oclIsKindOf(uml::Namespace) then res->union(m.
            oclAsType(uml::Namespace).getDependenciesRec) else res->
            union(Set{}) endif endif)

context Model

def: getDependencies : Set(Dependency) =
     self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{}
        | if m.oclIsTypeOf(uml::Dependency) then res->union(m.
       oclAsType(uml::Dependency)->asSet())
          else if m.oclIsKindOf(uml::Namespace) then res->union(m.
            oclAsType(uml::Namespace).getDependenciesRec) else res->
            union(Set{}) endif endif)

context Class

def: isContract : Boolean = self.getAppliedStereotypes()->select(
   name='contract')->size()<>0
def: isAssumption : Boolean = self.getAppliedStereotypes()->select
   (name='assumption')->size()<>0
def: itsAssumption : Class =
     self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class)
        and a.type.oclAsType(uml::Class).isAssumption)->at(1).type.
       oclAsType(uml::Class)
def: isGuarantee : Boolean = self.getAppliedStereotypes()->select(
  name='guarantee')->size()<>0
def: itsGuarantee : Class =
     self.ownedAttribute->select(a | a.type.oclIsKindOf(uml::Class)
        and a.type.oclAsType(uml::Class).isGuarantee)->at(1).type.
       oclAsType(uml::Class)
def: isObserver : Boolean = self.getAppliedStereotypes()->select(
  name='observer')->size()<>0
def: isComposite : Boolean = self.ownedAttribute->select(a | a.
  type.oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).
  name <> 'Timer')->size() <> 0

context Dependency

def: isImplementation : Boolean = self.getAppliedStereotypes()->
  select(name='contractImplementation')->size()<>0
def: implSource : Class = self.client->asOrderedSet()->at(1).
  oclAsType(uml::Class)
def: implTarget : Class = self.supplier->asOrderedSet()->at(1).
  oclAsType(uml::Class)

def: isUsage : Boolean = self.getAppliedStereotypes()->select(name
  ='contractUse')->size()<>0
def: useSource : Property = self.client->asOrderedSet()->at(1).
  oclAsType(uml::Property)
def: useTarget : Class = self.supplier->asOrderedSet()->at(1).
  oclAsType(uml::Class)
def: reqTarget : Class = self.getValue(self.getAppliedStereotypes
  ()->select(name='contractUse')->asOrderedSet()->at(1), 'reqTarget
  ').oclAsType(uml::Class)
def: refTarget : Class = self.getValue(self.getAppliedStereotypes
  ()->select(name='contractUse')->asOrderedSet()->at(1), 'refTarget
  ').oclAsType(uml::Class)

def: isConformance : Boolean = self.getAppliedStereotypes()->
  select(name='contractConformance')->size()<>0
def: confSource : Class = self.client->asOrderedSet()->at(1).
  oclAsType(uml::Class)
def: confTarget : Set(Class) = self.supplier.oclAsType(uml::Class)
  ->asSet()

context Property

```

```
def: isUsingContract(targetContract:Class, req:Class) : Boolean =
  self.clientDependency->select(d | d.isUsage and d.reqTarget = req
   and d.refTarget = targetContract)->size() > 0
def: getContractUseRelations : Set(Dependency) = self.
  clientDependency->select(d | d.isUsage)
def: isUsingContracts : Boolean = self.getContractUseRelations->
  size() > 0
def: isRefined(targetContract:Class, req:Class) : Boolean =
    self.type.oclIsTypeOf(uml::Class) and self.type.oclAsType(uml
      ::Class).isComposite and
        self.type.oclAsType(uml::Class).ownedAttribute->exists(a |
          a.type.oclIsTypeOf(uml::Class) and a.isComposite and a.
         isUsingContract(targetContract, req))

context Class

def: getPart : Set(Property) = self.ownedAttribute->select(a | a.
  type.oclIsTypeOf(uml::Class) and a.isComposite)
def: getUsedContractsOfParts(targetContract:Class, req:Class) : Set
  (Class) =
    self.getPart->iterate(p:Property; res:Set(Class)=Set{} | res->
      union(p.clientDependency->select(d | d.isUsage and
        d.reqTarget = req and d.refTarget = targetContract).
         useTarget).oclAsType(uml::Class))
def: getPortsFromUsedContractsOfParts(targetContract:Class, req:
  Class) : Set(Port) =
    self.getUsedContractsOfParts(targetContract, req)->iterate(c:
      Class; res:Set(Port)=Set{} | res->union(c.itsGuarantee.
      ownedPort))

context Port

def: interface : Interface = self.provided->asOrderedSet()->at(1).
  oclAsType(uml::Interface)
def: direction: String =
    if self.getValue(self.getAppliedStereotypes()->select(name='
      RhpPort')->asOrderedSet()->at(1), 'isReversed').oclAsType(
      Boolean)
        then 'required'
    else 'provided'
    endif
def: isIdenticalTo(p:Port) : Boolean = self.name = p.name and self
  .direction = p.direction and self.interface = p.interface
def: hasTypeIdenticalTo(p:Port) : Boolean = self.direction = p.
  direction and self.interface = p.interface
def: isConjugatedOf(p:Port) : Boolean = self.direction <> p.
  direction and self.interface = p.interface
```

```
-- context Assumption
context Class

-- Rule: An assumption has only properties with predefined types (
   i.e. an assumption is not involved in associations and
   aggregations)
def: assumptionHasNoPropertiesClassType : Boolean =
     self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class)
        and a.type.name<>'Timer' and not a.isComposite)->size() = 0

def: assumptionPropertiesWellFormed : Boolean =
     if self.isAssumption
         then self.assumptionHasNoPropertiesClassType
     else
         true
     endif


-- Rule: An assumption is not involved in any generalization
   relations (has no parents)
def: assumptionHasNoGenerals : Boolean =
     if self.isAssumption
         then self.general->size() = 0
     else
         true
     endif


-- Rule: An assumption does not depend on any model element
def: assumptionHasNoDependencies : Boolean =
     if self.isAssumption
         then self.clientDependency->reject(oclIsTypeOf(uml::
           InterfaceRealization))->size() = 0
     else
         true
     endif


inv assumptionWellFormed : self.assumptionPropertiesWellFormed and
    self.assumptionHasNoGenerals and self.
   assumptionHasNoDependencies
```

```
-- context Guarantee
context Class

-- Rule: A guarantee has only properties with predefined types (i.e. an assumption is not involved in associations and aggregations)
def: guaranteeHasNoPropertiesClassType : Boolean =
    self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.name<>'Timer' and not a.isComposite)->size() = 0

def: guaranteePropertiesWellFormed : Boolean =
    if self.isGuarantee
        then self.guaranteeHasNoPropertiesClassType
    else
        true
    endif


-- Rule: A guarantee is not involved in any generalization relations (has no parents)
def: guaranteeHasNoGenerals : Boolean =
    if self.isGuarantee
        then self.general->size() = 0
    else
        true
    endif


-- Rule: A guarantee does not depend on any model element
def: guaranteeHasNoDependencies : Boolean =
    if self.isGuarantee
        then self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization))->size() = 0
    else
        true
    endif


inv guaranteeWellFormed : self.guaranteePropertiesWellFormed and self.guaranteeHasNoGenerals and self.guaranteeHasNoDependencies



-- context Contract
context Class

```

```
-- Rule: A contract has no properties besides one part typed assumption and one part typed guarantee (i.e. no properties with predefined type and no properties
--        from associations, aggregations or compositions)
def: contractHasNoPropertiesPredefinedType : Boolean =
    self.ownedAttribute->select(a | not a.type.oclIsTypeOf(uml::Class))->size() = 0
def: contractHasNoPropertiesClassType : Boolean =
    self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and
        ((not a.type.oclAsType(uml::Class).isAssumption) or (a.type.oclAsType(uml::Class).isAssumption and not a.isComposite)) and
        ((not a.type.oclAsType(uml::Class).isGuarantee) or (a.type.oclAsType(uml::Class).isGuarantee and not a.isComposite)))->size() = 0
def: contractPropertiesWellFormed : Boolean =
    if self.isContract
        then self.contractHasNoPropertiesPredefinedType and self.contractHasNoPropertiesClassType
    else
        true
    endif


-- Rule: A contract has no operations
def: contractHasNoOperations : Boolean =
    if self.isContract
        then self.ownedOperation->size() = 0
    else
        true
    endif


-- Rule: A contract has no statemachine
def : contractHasNoStateMachine : Boolean =
    if self.isContract
        then self.ownedBehavior->size() = 0
    else
        true
    endif


-- Rule: A contract is not involved in any generalization relations (has no parents)
def : contractHasNoGenerals : Boolean =
    if self.isContract
```

```
        then self.general->size() = 0
    else
        true
    endif

inv contractWellFormed : self.contractPropertiesWellFormed and self.contractHasNoOperations and self.contractHasNoStateMachine and self.contractHasNoGenerals



-- context Contract
context Class

-- Rule: The assumption and guarantee of a contract define a closed system with respect to ports
def: haveIdenticalNoOfPorts : Boolean = self.itsAssumption.ownedPort->size() = self.itsGuarantee.ownedPort->size()
def: assumptionPortsSubsetGuaranteePorts : Boolean =
    self.itsAssumption.ownedPort->forAll(p1 | self.itsGuarantee.ownedPort->select(p2| p1.isConjugatedOf(p2))->size() >= 1)
def: guaranteePortsSubsetAssumptionPorts : Boolean =
    self.itsGuarantee.ownedPort->forAll(p1 | self.itsAssumption.ownedPort->select(p2 | p1.isConjugatedOf(p2))->size() >= 1)

def: contractAGPortsWellFormed : Boolean =
    if self.isContract
        then self.haveIdenticalNoOfPorts and self.assumptionPortsSubsetGuaranteePorts and self.guaranteePortsSubsetAssumptionPorts
    else
        true
    endif

inv contractClosedSystem : self.contractAGPortsWellFormed



-- context Implementation
context Dependency

-- Rule: The set of ports of the Guarantee is a subset or equal to the set of ports of the Part implementing it
-- Two ports are identical if they have the same name, direction and type
def: guaranteePortsSubsetPartPorts : Boolean =
    self.implTarget.itsGuarantee.ownedPort->forAll(p1 | self.implSource.ownedPort->select(p2 | p2.isIdenticalTo(p1))->size() = 1)

def: guaranteePortsWellFormed : Boolean =
    if self.isImplementation
        then self.guaranteePortsSubsetPartPorts
    else
        true
    endif

inv implementationGuaranteePortsWellFormed : self.guaranteePortsWellFormed



-- context ContractUse
context Dependency

def: getImplementationsForTarget : Set(Class) =
    let ut:Class = self.supplier->asOrderedSet()->at(1).oclAsType(uml::Class) in
    ut.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.isImplementation and
        d.supplier->asOrderedSet()->at(1).oclAsType(uml::Class) = ut).client.oclAsType(uml::Class)->asSet()

-- Rule: A contract can be used in a proof tree if and only if the type of the property using it implements the contract
def: canContractBeUsed : Boolean = self.getImplementationsForTarget->includes(self.client->asOrderedSet()->at(1).oclAsType(uml::Property).type.oclAsType(uml::Class))

def: contractUseWellFormed : Boolean =
    if self.isUsage
        then self.canContractBeUsed
    else
        true
    endif

inv contractUseWellFormed : self.contractUseWellFormed



context Property


```

```
def: refTargetPortsSubsetSourcesPorts(targetContract:Class, req:Class) : Boolean =
    let sp:Set(Port) = self.type.oclAsType(uml::Class).getPortsFromUsedContractsOfParts(targetContract,req) in
        targetContract.itsGuarantee.ownedPort->forAll(p1 | sp->select(p2 | p1.hasTypeIdenticalTo(p2))->size() >= 1)

-- Rule: The set of ports of a dominated guarantee have a correspondent within the set of ports of the dominating gurantees
def: targetGuaranteePortsWellFormed : Boolean =
    self.getContractUseRelations->forAll(d |
        if self.isRefined(d.useTarget,d.reqTarget)
            then self.refTargetPortsSubsetSourcesPorts(d.useTarget, d.reqTarget)
        else
            true
        endif)

inv refinementTargetGuaranteePortsWellFormed : self.targetGuaranteePortsWellFormed



context Property

-- Rule: A component can use at most one contract for the satisfaction of one requirement and within one dominance
def: isContractUniqueForRequirementAndRefinement : Boolean =
    let r:Set(Dependency) = self.getContractUseRelations in
    if self.type.oclIsTypeOf(uml::Class) and self.isUsingContracts
        then r->forAll(d1 | if r->excluding(d1)->select(d2 | d1.reqTarget = d2.reqTarget)->size() = 0 then true else
                    r->select(d2 | d1.reqTarget = d2.reqTarget)->size() =
                    r->select(d2 | d1.reqTarget = d2.reqTarget).refTarget->asSet()->size() endif)
    else
        true
    endif

inv contractUseUniqueRR : self.isContractUniqueForRequirementAndRefinement



-- context SafetyProperty
context Class
```

```
-- Rule: All SafetyProperties have a contract conforming to it
def: isVerified : Boolean =
    self.oclAsType(uml::Classifier).getModel().getDependencies->
      select(d | d.isConformance and d.confTarget->includes(self))
     ->size() > 0

def: spIsVerified : Boolean =
    if self.isObserver
        then self.isVerified
    else
        true
    endif

inv safetyPropertyIsVerified : self.spIsVerified
```

A.2 Rules Defined on the OMEGA Contracts Profile for Enforcing the Meta-Model

```
MainModel : http://www.eclipse.org/uml2/3.0.0/UML

context Class

def: isContract : Boolean = self.getAppliedStereotypes()->select(
  name='contract')->size()<>0
def: isAssumption : Boolean = self.getAppliedStereotypes()->select
  (name='assumption')->size()<>0
def: isGuarantee : Boolean = self.getAppliedStereotypes()->select(
  name='guarantee')->size()<>0
def: isObserver : Boolean = self.getAppliedStereotypes()->select(
  name='observer')->size()<>0
def: isNotStereotypedClass : Boolean = not self.isContract and not
   self.isAssumption and not self.isGuarantee and not self.
  isObserver
--def: isComposite : Boolean = self.ownedAttribute->select(a | a.
  type.oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).
  name <> 'Timer')->size() <> 0

context Dependency

def: isConformance : Boolean = self.getAppliedStereotypes()->
  select(name='contractConformance')->size()<>0
def: isImplementation : Boolean = self.getAppliedStereotypes()->
  select(name='contractImplementation')->size()<>0
```

```
def: isUsage : Boolean = self.getAppliedStereotypes()->select(name='contractUse')->size()<>0


context Class

-- Rule: Stereotypes are disjoint
inv correctDefinition : (isContract and not isAssumption and not isGuarantee and not isObserver) or
                        (not isContract and isAssumption and not isGuarantee and not isObserver) or
                        (not isContract and not isAssumption and isGuarantee and not isObserver) or
                        (not isContract and not isAssumption and not isGuarantee and isObserver) or
                        (not isContract and not isAssumption and not isGuarantee and not isObserver)

-- Rule: A contract has one composite assumption and guarantee
def: contractHasAssumption : Boolean =
    self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).isAssumption and a.isComposite)->size() = 1
def: contractHasGuarantee : Boolean =
    self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).isGuarantee and a.isComposite)->size() = 1
def: contractPropertiesWellFormed : Boolean =
    if self.isContract
        then self.contractHasAssumption and self.contractHasGuarantee
    else
        true
    endif

inv contractPropertiesWellFormed: self.contractPropertiesWellFormed


context Dependency

-- Rule: Stereotypes are disjoint
inv correctDefinition : (isImplementation and not isUsage and not isConformance) or
                        (not isImplementation and isUsage and not isConformance) or
                        (not isImplementation and not isUsage and isConformance) or
                        (not isImplementation and not isUsage and not isConformance)

-- Rule: The source of a Conformance relation is a Contract and its cardinality equals to 1
def: conformanceHas1Source : Boolean =
    self.client->size() = 1
def: conformanceHasContractSource : Boolean =
    let s:NamedElement = self.client->asOrderedSet()->at(1) in
    s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isContract
def: conformanceSourceWellFormed : Boolean =
    if self.isConformance
        then self.conformanceHas1Source and self.conformanceHasContractSource
    else
        true
    endif

inv conformanceSourceWellFormed : self.conformanceSourceWellFormed

-- Rule: The target of a Conformance relation is a set of Observers
def: conformanceHasTargets : Boolean =
    self.supplier->size() > 0
def: conformanceHasObserverTargets : Boolean =
    self.supplier->forAll(s | s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isObserver)
def: conformanceTargetWellFormed : Boolean =
    if self.isConformance
        then self.conformanceHasTargets and self.conformanceHasObserverTargets
    else
        true
    endif

inv conformanceTargetWellFormed : self.conformanceTargetWellFormed

-- Rule: The source of an Implementation relation is a Class and its cardinality equals to 1
def: implementationHas1Source : Boolean =
    self.client->size() = 1
def: implementationHasClassSource : Boolean =
    let s:NamedElement = self.client->asOrderedSet()->at(1) in
    s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isNotStereotypedClass
def: implementationSourceWellFormed : Boolean =
    if self.isImplementation
        then self.implementationHas1Source and self.implementationHasClassSource
    else
        true
    endif

inv implementationSourceWellFormed : self.implementationSourceWellFormed

-- Rule: The target of an Implementation relation is a Contract and its cardinality equals to 1
def: implementationHas1Target : Boolean =
    self.supplier->size() = 1
def: implementationHasContractTarget : Boolean =
    let s:NamedElement = self.supplier->asOrderedSet()->at(1) in
    s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isContract
def: implementationTargetWellFormed : Boolean =
    if self.isImplementation
        then self.implementationHas1Target and self.implementationHasContractTarget
    else
        true
    endif

inv implementationTargetWellFormed : self.implementationTargetWellFormed

-- Rule: The source of a ContractUse relation is a Property and its cardinality equals to 1
def: usageHas1Source : Boolean =
    self.client->size() = 1
def: usageHasPropertySource : Boolean =
    let s:NamedElement = self.client->asOrderedSet()->at(1) in
    s.oclIsTypeOf(uml::Property) and not s.oclIsTypeOf(uml::Port)
def: usageSourceWellFormed : Boolean =
    if self.isUsage
        then self.usageHas1Source and self.usageHasPropertySource
    else
        true
    endif

inv usageSourceWellFormed : self.usageSourceWellFormed
```

```

-- Rule: The target of a ContractUse relation is a Contract and its cardinality equals to 1
 def: usageHas1Target : Boolean =
     self.supplier->size() = 1
 def: usageHasContractTarget : Boolean =
     let s:NamedElement = self.supplier->asOrderedSet()->at(1) in
     s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isContract
 def: usageTargetWellFormed : Boolean =
     if self.isUsage
         then self.usageHas1Target and self.usageHasContractTarget
     else
         true
     endif

 inv usageTargetWellFormed : self.usageTargetWellFormed

-- Rule: The requirement for a ContractUse relation is an Observer and its cardinality equals to 1
 def: usageHas1Requirement : Boolean =
     self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'reqTarget')->size() = 1
 def: usageHasObserverRequirement : Boolean =
     self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'reqTarget').oclIsTypeOf(uml::Class) and
     self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'reqTarget').oclAsType(uml::Class).isObserver
 def: usageReqTargetWellFormed : Boolean =
     if self.isUsage
         then self.usageHas1Requirement and self.usageHasObserverRequirement
     else
         true
     endif

 inv usageReqTargetWellFormed: self.usageReqTargetWellFormed

-- Rule: The dominated contract for a ContractUse relation if defined is a Contract
 def: usageHasNoDominatedContract : Boolean =
     self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'refTarget')->size() = 0
```

```
def : usageHas1DominatedContract : Boolean =
    self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'refTarget')->size() = 1
def: usageHasCorrectDominatedContract : Boolean =
    self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'refTarget').oclIsTypeOf(uml::Class) and
    self.getValue(self.getAppliedStereotypes()->select(name='contractUse')->asOrderedSet()->at(1), 'refTarget').oclAsType(uml::Class).isContract
def: usageDominatedContractWellFormed : Boolean =
    if self.isUsage
        then self.usageHasNoDominatedContract or (self.usageHas1DominatedContract and self.usageHasCorrectDominatedContract)
    else
        true
    endif

inv usageDominatedContractWellFormed: self.usageDominatedContractWellFormed
```

B Proofs of the Required Compositionality Results

The following theorems from the algebra of sets are used for the set computations in the proofs of compositionality properties:

1. $A \setminus A = \emptyset$
2. $A \setminus \emptyset = A$
3. $\emptyset \setminus A = \emptyset$
4. $B \setminus (A \setminus B) = B$
5. $(A \setminus B) \setminus A = \emptyset$
6. $A \setminus (A \setminus B) = A \cap B$
7. $(A \setminus B) \setminus B = A \setminus B$
8. $(A \setminus B) \setminus C = A \setminus (B \cup C)$
9. $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$
10. $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
11. $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$
12. $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$
13. $(A \cap B) \setminus C = A \cap (B \setminus C) = (A \setminus C) \cap B$
14. $(A \setminus B) \cup C = (A \cup C) \setminus (B \setminus C)$
15. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
16. $A \cup (A \cap B) = A$
17. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
18. $A \cap (A \cup B) = A$

where A, B and C are sets.

B.1 Proof of Theorem 6.1

Theorem 6.1. $(\mathcal{A}, \|)$ *is a commutative monoid, where* $\mathcal{A}$ *denotes the set of TIOA.*

Proof. Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ be three timed input/output automata.

1. *Commutativity*: $\mathcal{A}_1 \parallel \mathcal{A}_2 = \mathcal{A}_2 \parallel \mathcal{A}_1$ is true since the composition operator does not define an order at computation.
2. *Associativity*: By applying the composition operator we obtain $(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3 = \mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3) = (X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where
   - $X = X_1 \cup X_2 \cup X_3$.
   - $Clk = Clk_1 \cup Clk_2 \cup Clk_3$.
   - $Q = \{\mathrm{x}_1 \cup \mathrm{x}_2 \cup \mathrm{x}_3 | \mathrm{x}_1 \in Q_1, \mathrm{x}_2 \in Q_2 \text{ and } \mathrm{x}_3 \in Q_3\}$.
   - $\theta = \theta_1 \cup \theta_2 \cup \theta_3$.
   - $I = (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup (I_3 \setminus (O_1 \cup O_2))$.

     $I_{(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3} =$

     $= (I_{\mathcal{A}_1 \parallel \mathcal{A}_2} \setminus O_3) \cup (I_3 \setminus O_{\mathcal{A}_1 \parallel \mathcal{A}_2})$

     $= (((I_1 \setminus O_2) \cup (I_2 \setminus O_1)) \setminus O_3) \cup (I_3 \setminus ((O_1 \setminus I_2) \cup (O_2 \setminus I_1)$

     $= ((I_1 \setminus O_2) \setminus O_3) \cup ((I_2 \setminus O_1) \setminus O_3) \cup ((I_3 \setminus (O_1 \setminus I_2)) \cap (I_3 \setminus (O_2 \setminus I_1)))$

     $= (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup (((I_3 \cap I_2) \cup (I_3 \setminus O_1)) \cap ((I_3 \cap I_1) \cup (I_3 \setminus O_2)))$

     $= (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup ((I_3 \setminus O_1) \cap (I_3 \setminus O_2))$

     $= (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup (I_3 \setminus (O_1 \cup O_2))$

     $I_{\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3)} =$

     $= (I_1 \setminus O_{\mathcal{A}_2 \parallel \mathcal{A}_3}) \cup (I_{\mathcal{A}_2 \parallel \mathcal{A}_3} \setminus O_1)$

     $= (I_1 \setminus ((O_2 \setminus I_3) \cup (O_3 \setminus I_2))) \cup (((I_2 \setminus O_3) \cup (I_3 \setminus O_2)) \setminus O_1)$

     $= ((I_1 \setminus (O_2 \setminus I_3)) \cap (I_1 \setminus (O_3 \setminus I_2))) \cup (((I_2 \setminus O_3) \setminus O_1) \cup ((I_3 \setminus O_2) \setminus O_1))$

     $= (((I_1 \cap I_3) \cup (I_1 \setminus O_2)) \cap ((I_1 \cap I_2) \cup (I_1 \setminus O_3))) \cup ((I_2 \setminus (O_1 \cup O_3)) \cup ((I_3 \setminus (O_1 \cup O_2)))$

     $= ((I_1 \setminus O_2) \cap (I_1 \setminus O_3)) \cup ((I_2 \setminus (O_1 \cup O_3)) \cup ((I_3 \setminus (O_1 \cup O_2)))$

     $= (I_1 \setminus (O_2 \cup O_3)) \cup (I_2 \setminus (O_1 \cup O_3)) \cup (I_3 \setminus (O_1 \cup O_2))$
   - $O = (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup (O_3 \setminus (I_1 \cup I_3))$.

     $O_{(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3} =$

     $= (O_{\mathcal{A}_1 \parallel \mathcal{A}_2} \setminus I_3) \cup (O_3 \setminus I_{\mathcal{A}_1 \parallel \mathcal{A}_2})$

     $= (((O_1 \setminus I_2) \cup (O_2 \setminus I_1)) \setminus I_3) \cup (O_3 \setminus ((I_1 \setminus O_2) \cup (I_2 \setminus O_1)$

     $= ((O_1 \setminus I_2) \setminus I_3) \cup ((O_2 \setminus I_1) \setminus I_3) \cup ((O_3 \setminus (I_1 \setminus O_2)) \cap (O_3 \setminus (I_2 \setminus O_1)))$

     $= (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup (((O_3 \cap O_2) \cup (O_3 \setminus I_1)) \cap ((O_3 \cap O_1) \cup (O_3 \setminus I_2)))$

     $= (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup ((O_3 \setminus I_1) \cap (O_3 \setminus I_2))$

     $= (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup (O_3 \setminus (I_1 \cup I_2))$

     $O_{\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3)} =$

     $= (O_1 \setminus I_{\mathcal{A}_2 \parallel \mathcal{A}_3}) \cup (O_{\mathcal{A}_2 \parallel \mathcal{A}_3} \setminus I_1)$

     $= (O_1 \setminus ((I_2 \setminus O_3) \cup (I_3 \setminus O_2))) \cup (((O_2 \setminus I_3) \cup (O_3 \setminus I_2)) \setminus I_1)$

     $= ((O_1 \setminus (I_2 \setminus O_3)) \cap (O_1 \setminus (I_3 \setminus O_2))) \cup (((O_2 \setminus I_3) \setminus I_1) \cup ((O_3 \setminus I_2) \setminus I_1))$

$= (((O_1 \cap O_3) \cup (O_1 \setminus I_2)) \cap ((O_1 \cap O_2) \cup (O_1 \setminus I_3))) \cup ((O_2 \setminus (I_1 \cup I_3)) \cup$
$((O_3 \setminus (I_1 \cup I_2)))$
$= ((O_1 \setminus I_2) \cap (O_1 \setminus I_3)) \cup ((O_2 \setminus (I_1 \cup I_3)) \cup ((O_3 \setminus (I_1 \cup I_2)))$
$= (O_1 \setminus (I_2 \cup I_3)) \cup (O_2 \setminus (I_1 \cup I_3)) \cup (O_3 \setminus (I_1 \cup I_2))$

- $V = V_1 \cup V_2 \cup V_3 \cup (O_1 \cap (I_2 \cup I_3)) \cup (O_2 \cap (I_1 \cup I_3)) \cup (O_3 \cap (I_1 \cup I_2))$.
  $V_{(\mathcal{A}_1 \| \mathcal{A}_2) \| \mathcal{A}_3} =$
  $= V_{\mathcal{A}_1 \| \mathcal{A}_2} \cup V_3 \cup (O_{\mathcal{A}_1 \| \mathcal{A}_2} \cap I_3) \cup (I_{\mathcal{A}_1 \| \mathcal{A}_2} \cap O_3)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap I_2) \cup (O_2 \cap I_1) \cup (((O_1 \setminus I_2) \cup (O_2 \setminus I_1)) \cap I_3) \cup$
  $(((I_1 \setminus O_2) \cup (I_2 \setminus O_1)) \cap O_3)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap I_2) \cup (O_2 \cap I_1) \cup ((O_1 \setminus I_2) \cap I_3) \cup ((O_2 \setminus I_1) \cap$
  $I_3) \cup ((I_1 \setminus O_2) \cap O_3) \cup ((I_2 \setminus O_1) \cap O_3)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap I_2) \cup (O_2 \cap I_1) \cup (O_1 \cap (I_3 \setminus I_2)) \cup (O_2 \cap (I_3 \setminus I_1)) \cup$
  $(I_1 \cap (O_3 \setminus O_2)) \cup (I_2 \cap (O_3 \setminus O_1))$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap I_2) \cup (O_2 \cap I_1) \cup (O_1 \cap I_3) \cup (O_2 \cap I_3) \cup (I_1 \cap O_3) \cup (I_2 \cap O_3)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap (I_2 \cup I_3)) \cup (O_2 \cap (I_1 \cup I_3)) \cup (O_3 \cap (I_1 \cup I_2))$
  $V_{\mathcal{A}_1 \| (\mathcal{A}_2 \| \mathcal{A}_3)} =$
  $= V_1 \cup V_{\mathcal{A}_2 \| \mathcal{A}_3} \cup ((O_1 \cap I_{\mathcal{A}_2 \| \mathcal{A}_3}) \cup (O_{\mathcal{A}_2 \| \mathcal{A}_3} \cap I_1))$
  $= V_1 \cup V_2 \cup V_3 \cup (O_2 \cap I_3) \cup (I_2 \cap O_3) \cup (((I_2 \setminus O_3) \cup (I_3 \setminus O_2)) \cap O_1) \cup$
  $(((O_2 \setminus I_3) \cup (O_3 \setminus I_2)) \cap I_1)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_2 \cap I_3) \cup (I_2 \cap O_3) \cup ((I_2 \setminus O_3) \cap O_1) \cup ((I_3 \setminus O_2) \cap$
  $O_1) \cup ((O_2 \setminus I_3) \cap I_1) \cup ((O_3 \setminus I_2) \cap I_1)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_2 \cap I_3) \cup (I_2 \cap O_3) \cup (I_2 \cap (O_1 \setminus O_3)) \cup (I_3 \cap (O_1 \setminus$
  $O_2)) \cup (O_2 \cap (I_1 \setminus I_3)) \cup (O_3 \cap (I_1 \setminus I_2))$
  $= V_1 \cup V_2 \cup V_3 \cup (O_2 \cap I_3) \cup (I_2 \cap O_3) \cup (I_2 \cap O_1) \cup (I_3 \cap O_1) \cup (O_2 \cap I_1) \cup (O_3 \cap I_1)$
  $= V_1 \cup V_2 \cup V_3 \cup (O_1 \cap (I_2 \cup I_3)) \cup (O_2 \cap (I_1 \cup I_3)) \cup (O_3 \cap (I_1 \cup I_2))$
- $H = H_1 \cup H_2 \cup H_3$.
- D is the set of discrete transitions where for each $\mathrm{x} = \mathrm{x}_1 \cup \mathrm{x}_2 \cup \mathrm{x}_3$ and $\mathrm{x}' = \mathrm{x}'_1 \cup \mathrm{x}'_2 \cup \mathrm{x}'_3 \in Q$ and each $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ if and only if for $i \in \{1, 2, 3\}$, either
  (a) $a \in A_i$ and $\mathrm{x}_i \xrightarrow{a} \mathrm{x}'_i$, or
  (b) $a \notin A_i$ and $\mathrm{x}_i = \mathrm{x}'_i$.
- $\tau \in \mathcal{T} \Leftrightarrow \tau \lceil X_i \in \mathcal{T}_i$, $i \in \{1, 2, 3\}$.

3. The identity element is the *empty* timed input/output automaton: it has no internal variables, it does not perform any actions and can let time elapse to infinity.

□

B.2 Proof of Proposition 6.1

Proposition 6.1. *Let K_1, K_2, K_3 be three components* not necessarily comparable *and Env an environment such that $K_1 \sqsubseteq_{Env} K_2$ and $K_2 \sqsubseteq_{Env} K_3$. Then $K_1 \sqsubseteq_{Env} K_3$.*

Proof. $K_1 \sqsubseteq_{Env} K_2 \overset{\Delta}{\Leftrightarrow} K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K'_2 \parallel Env'$ (1)

We write the automaton $Env' = Env'_1 \parallel Env'_2$ where

- $Env'_1 = (\emptyset, \{\phi\}, \phi, ((O_{K_1} \cap O_{K_2}) \setminus I_{Env}), ((I_{K_1} \cap I_{K_2}) \setminus O_{Env}), \emptyset, \emptyset, D_{Env'_1}, 2_0^{[\mathbb{R}+]})$
- $Env'_2 = (\emptyset, \{\phi\}, \phi, ((O_{K_1} \setminus O_{K_2}) \setminus I_{Env}), ((I_{K_1} \setminus I_{K_2}) \setminus O_{Env}), \emptyset, \emptyset, D_{Env'_2}, 2_0^{[\mathbb{R}+]})$

Remark that the sets of input and output actions are pairwise disjoint for Env'_1 and Env'_2.

We write the automaton $K'_2 = K''_2 \parallel Env'_3$ where

- $K''_2 = (\emptyset, \{\phi\}, \phi, (I_{K_1} \setminus I_{K_2}), (O_{K_1} \setminus O_{K_2}), (V_{K_1} \setminus E_{K_2}), \emptyset, D_{K''_2}, 2_0^{[\mathbb{R}+]})$
- $Env'_3 = (\emptyset, \{\phi\}, \phi, (V_{K_1} \cap O_{K_2}), (V_{K_1} \cap I_{K_2}), \emptyset, \emptyset, D_{Env'_3}, 2_0^{[\mathbb{R}+]})$

Similarly, the sets of input, output and visible actions are pairwise disjoint for K''_2 and Env'_3.

With this notation:
(1) $\Leftrightarrow K_1 \parallel Env \parallel Env'_1 \parallel Env'_2 \preceq K_2 \parallel Env \parallel K''_2 \parallel Env'_3 \parallel Env'_1 \parallel Env'_2$ (2)

$K_2 \sqsubseteq_{Env} K_3 \overset{\Delta}{\Leftrightarrow} K_2 \parallel Env \parallel Env'' \preceq K_3 \parallel Env \parallel K'_3 \parallel Env''$ (3)

With the same notation we obtain that $Env'' = Env'_1 \parallel Env'_3$, and
(3) $\Leftrightarrow K_2 \parallel Env \parallel Env'_1 \parallel Env'_3 \preceq K_3 \parallel Env \parallel K'_3 \parallel Env'_1 \parallel Env'_3$ (4)

Indeed, the Env'' and $Env'_1 \parallel Env'_3$ have the same signature and the same structure of the automata:

- $I_{Env''} = ((O_{K_1} \cap O_{K_2}) \setminus I_{Env}) \cup (V_{K_1} \cap O_{K_2}) = ((O_{K_1} \cap O_{K_2}) \cup (V_{K_1} \cap O_{K_2})) \setminus (I_{Env} \setminus (V_{K_1} \cap O_{K_2})) = ((V_{K_1} \cup O_{K_1}) \cap O_{K_2}) \setminus I_{Env} = O_{K_2} \setminus I_{Env}$
- $O_{Env''} = ((I_{K_1} \cap I_{K_2}) \setminus O_{Env}) \cup (V_{K_1} \cap I_{K_2}) = ((I_{K_1} \cap I_{K_2}) \cup (V_{K_1} \cap I_{K_2})) \setminus (O_{Env} \setminus (V_{K_1} \cap I_{K_2})) = ((V_{K_1} \cup I_{K_1}) \cap I_{K_2}) \setminus O_{Env} = I_{K_2} \setminus O_{Env}$
- $V_{Env''} = \emptyset$

Composing (4) with $K''_2 \parallel Env'_2$ and from Theorem 1.1 we get:
$K_2 \parallel Env \parallel Env'_1 \parallel Env'_3 \parallel K''_2 \parallel Env'_2 \preceq K_3 \parallel Env \parallel K'_3 \parallel Env'_1 \parallel Env'_3 \parallel K''_2 \parallel Env'_2 \Leftrightarrow$

$$\left.\begin{array}{r}\Leftrightarrow K_2 \parallel Env \parallel K_2'' \parallel Env_3' \parallel Env_1' \parallel Env_2' \preceq K_3 \parallel Env \parallel K_3' \parallel K_2' \parallel Env_1' \parallel Env_2' \\ (2)\ K_1 \parallel Env \parallel Env_1' \parallel Env_2' \preceq K_2 \parallel Env \parallel K_2'' \parallel Env_3' \parallel Env_1' \parallel Env_2'\end{array}\right\} \overset{\mathit{Transitivity\ of}}{\Longrightarrow}$$

$$\Longrightarrow\ K_1 \parallel Env \parallel Env_1' \parallel Env_2' \preceq K_3 \parallel Env \parallel K_3' \parallel K_2' \parallel Env_1' \parallel Env_2' \Leftrightarrow$$
$$\Leftrightarrow K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel K_2' \parallel K_3' \parallel Env'$$

By denoting $K' = K_2' \parallel K_3'$ we have:
$K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel K' \parallel Env' \overset{\Delta}{\Leftrightarrow} K_1 \sqsubseteq_{Env} K_3$

We prove that K' is indeed the automaton generated by the refinement under context relation. Since K_2' and K_3' are built from the hypothesis by the refinement under context relation, by composition they define the correct structure for K'. Moreover:

- $I_{K_2'\|K_3'} = (I_{K_2'} \setminus O_{K_3'}) \cup (I_{K_3'} \setminus O_{K_2'}) =$
  $= (((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})) \setminus ((O_{K_2} \setminus O_{K_3}) \cup (V_{K_2} \cap I_{K_3}))) \cup (((I_{K_2} \setminus I_{K_3}) \cup$
  $(V_{K_2} \cap O_{K_3})) \setminus ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2})))$
  $= ((((I_{K_1} \setminus I_{K_2}) \setminus (O_{K_2} \setminus O_{K_3})) \cap ((I_{K_1} \setminus I_{K_2}) \setminus (V_{K_2} \cap I_{K_3}))) \cup (((V_{K_1} \cap O_{K_2}) \setminus$
  $(O_{K_2} \setminus O_{K_3})) \cap ((V_{K_1} \cap O_{K_2}) \setminus (V_{K_2} \cap I_{K_3})))) \cup ((((I_{K_2} \setminus I_{K_3}) \setminus (O_{K_1} \setminus O_{K_2})) \cap ((I_{K_2} \setminus$
  $I_{K_3}) \setminus (V_{K_1} \cap I_{K_2}))) \cup (((V_{K_2} \cap O_{K_3}) \setminus (O_{K_1} \setminus O_{K_2})) \cap ((V_{K_2} \cap O_{K_3}) \setminus (V_{K_1} \cap I_{K_2}))))$
  $= (((((I_{K_1} \setminus I_{K_2}) \setminus O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{K_3})) \cap (((I_{K_1} \setminus I_{K_2}) \setminus V_{K_2}) \cup ((I_{K_1} \setminus$
  $I_{K_2}) \setminus I_{K_3}))) \cup ((V_{K_1} \cap (O_{K_2} \setminus (O_{K_2} \setminus O_{K_3}))) \cap (((V_{K_1} \cap O_{K_2}) \setminus V_{K_2}) \cup ((V_{K_1} \cap$
  $O_{K_2}) \setminus I_{K_3})))) \cup (((((I_{K_2} \setminus I_{K_3} \setminus O_{K_1}) \cup ((I_{K_2} \setminus I_{K_3}) \cap O_{K_2})) \cap (((I_{K_2} \setminus I_{K_3}) \setminus$
  $V_{K_1}) \cup ((I_{K_2} \setminus I_{K_3}) \setminus I_{K_2}))) \cup ((O_{K_3} \cap (V_{K_2} \setminus (O_{K_1} \setminus O_{K_2}))) \cap (((V_{K_1} \cap O_{K_3}) \setminus$
  $V_{K_1}) \cup ((V_{K_2} \cap O_{K_3}) \setminus I_{K_2}))))$
  $= ((((I_{K_1} \setminus (I_{K_2} \cup O_{K_2})) \cup ((I_{K_1} \cap O_{K_3}) \setminus I_{K_2})) \cap ((I_{K_1} \setminus (I_{K_2} \cup V_{K_2})) \cup ((I_{K_1} \setminus$
  $I_{K_2}) \setminus I_{K_3}))) \cup ((V_{K_1} \cap O_{K_2} \cap O_{K_3}) \cap ((V_{K_1} \cap (O_{K_2} \setminus V_{K_2})) \cup (V_{K_1} \cap (O_{K_2} \setminus I_{K_3}))))) \cup$
  $((((I_{K_2} \setminus (I_{K_3} \cup O_{K_1})) \cup ((I_{K_2} \cap O_{K_2}) \setminus I_{K_3})) \cap ((I_{K_2} \setminus (I_{K_3} \cup V_{K_1})) \cup \emptyset)) \cup ((O_{K_3} \cap$
  $((V_{K_2} \setminus O_{K_1}) \cup (V_{K_2} \cap O_{K_2}))) \cap ((V_{K_1} \setminus V_{K_1}) \cap O_{K_3}) \cup ((V_{K_2} \setminus I_{K_2}) \cap O_{K_3}))))$
  $= (((((I_{K_1} \setminus I_{K_2}) \cap (I_{K_1} \setminus O_{K_2})) \cup \setminus) \cap (((I_{K_1} \setminus V_{K_2}) \setminus I_{K_2}) \cup (I_{K_1} \setminus (I_{K_2} \cup I_{K_3})))) \cup$
  $((V_{K_1} \cap O_{K_2} \cap O_{K_3}) \cap ((V_{K_1} \cap O_{K_2}) \cup (V_{K_1} \cap O_{K_2})))) \cup (((((I_{K_2} \setminus I_{K_3}) \cap (I_{K_2} \setminus$
  $O_{K_1})) \cup \emptyset) \cap ((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1})) \cup ((O_{K_3} \cap (V_{K_2} \setminus O_{K_1})) \cap (\emptyset \cup (V_{K_2} \cap O_{K_3}))))$
  $= (((((I_{K_1} \setminus I_{K_2}) \cap I_{K_1}) \cup \emptyset) \cap ((I_{K_1} \setminus I_{K_2}) \cup (I_{K_1} \setminus (I_{K_2} \cup I_{K_3})))) \cup ((V_{K_1} \cap$
  $O_{K_2} \cap O_{K_3}) \cap (V_{K_1} \cap O_{K_2}))) \cup ((((I_{K_2} \setminus I_{K_3}) \cap I_{K_2}) \cap ((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1})) \cup$
  $((V_{K_2} \cap (O_{K_3} \setminus O_{K_1})) \cap (V_{K_2} \cap O_{K_3})))$
  $= (((I_{K_1} \setminus I_{K_3}) \cap ((I_{K_1} \setminus I_{K_2}) \cup (I_{K_1} \setminus (I_{K_2} \cup I_{K_3})))) \cup ((V_{K_1} \cap O_{K_2} \cap O_{K_3}) \cap (V_{K_1} \cap$
  $O_{K_2}))) \cup (((I_{K_2} \setminus I_{K_3}) \cap ((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1})) \cup ((V_{K_2} \cap (O_{K_3} \setminus O_{K_1})) \cap (V_{K_2} \cap O_{K_3})))$
  $= ((((I_{K_1} \setminus I_{K_2}) \cap (I_{K_1} \setminus I_{K_2})) \cup ((I_{K_1} \setminus I_{K_2}) \cap ((I_{K_1} \setminus I_{K_2}) \setminus I_{K_3}))) \cup (V_{K_1} \cap O_{K_2} \cap$
  $O_{K_3})) \cup (((I_{K_2} \setminus I_{K_3}) \cap ((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1})) \cup (((V_{K_2} \cap O_{K_3}) \setminus O_{K_1}) \cap (V_{K_2} \cap O_{K_3})))$
  $= (((I_{K_1} \setminus I_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \setminus I_{K_3})) \cup (V_{K_1} \cap O_{K_2} \cap O_{K_3})) \cup (((I_{K_2} \setminus I_{K_3}) \setminus$

$V_{K_1}) \cup ((V_{K_2} \cap O_{K_3}) \setminus O_{K_1}))$
$= (((I_{K_1} \setminus I_{K_2}) \setminus (I_{K_3} \setminus (I_{K_1} \setminus I_{K_2}))) \cup (V_{K_1} \cap O_{K_2} \cap O_{K_3})) \cup (((I_{K_2} \setminus I_{K_3}) \setminus$
$V_{K_1}) \cup (V_{K_2} \cap (O_{K_3} \setminus O_{K_1})))$
$= ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2} \cap O_{K_3})) \cup (((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1}) \cup (V_{K_2} \cap (O_{K_3} \setminus O_{K_1})))$
$= (I_{K_1} \setminus I_{K_2}) \cup ((I_{K_2} \setminus I_{K_3}) \setminus V_{K_1}) \cup (V_{K_1} \cap O_{K_2} \cap O_{K_3}) \cup (V_{K_2} \cap (O_{K_3} \setminus O_{K_1}))$
$= ((I_{K_1} \cup ((I_{K_2} \setminus V_{K_1}) \setminus I_{K_3})) \setminus (I_{K_2} \setminus ((I_{K_2} \setminus V_{K_1}) \setminus I_{K_3}))) \cup (((V_{K_1} \cap O_{K_3}) \cap$
$O_{K_2}) \cup ((V_{K_2} \cap O_{K_3}) \setminus O_{K_1}))$
$= (((I_{K_1} \cup (I_{K_2} \setminus V_{K_1})) \setminus (I_{K_3} \setminus I_{K_1})) \setminus ((I_{K_2} \cap I_{K_3}) \cup (I_{K_2} \setminus (I_{K_2} \setminus V_{K_1})))) \cup$
$(((V_{K_1} \cap O_{K_3}) \cup ((V_{K_1} \cap O_{K_3}) \cap O_{K_2})) \setminus (O_{K_1} \setminus ((V_{K_1} \cap O_{K_3}) \cap O_{K_2})))$
$= ((((I_{K_1} \cup I_{K_2}) \setminus (V_{K_1} \setminus I_{K_1})) \setminus (I_{K_3} \setminus I_{K_1})) \setminus ((I_{K_2} \cap I_{K_3}) \cup (I_{K_2} \cap V_{K_1}))) \cup ((((V_{K_1} \cap$
$O_{K_3}) \cup (V_{K_2} \cap O_{K_3})) \cap ((V_{K_2} \cap O_{K_3}) \cup O_{K_2}))) \setminus ((O_{K_1} \setminus (V_{K_1} \cap O_{K_3}) \cup (O_{K_1} \setminus O_{K_2})))$
$= ((((I_{K_1} \cup I_{K_2}) \setminus V_{K_1}) \setminus (I_{K_3} \setminus I_{K_1}) \setminus (I_{K_2} \cap (I_{K_3} \cup V_{K_1}))) \cup ((((V_{K_1} \cup V_{K_2}) \cap$
$O_{K_3}) \cap ((V_{K_2} \cap O_{K_3}) \cup O_{K_2}))) \setminus (O_{K_1} \cup (O_{K_1} \setminus O_{K_2})))$
$= ((I_{K_1} \setminus (I_{K_3} \setminus I_{K_1})) \setminus (I_{K_2} \cap (I_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap O_{K_3}) \cap ((V_{K_2} \cap O_{K_3}) \cup$
$O_{K_2})) \setminus (O_{K_1} \setminus (O_{K_2} \setminus O_{K_1})))$
$= (I_{K_1} \setminus (I_{K_2} \cap (I_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap O_{K_3} \cap V_{K_2}) \cup (V_{K_1} \cap O_{K_3} \cap O_{K_2})) \setminus O_{K_1})$
$= ((I_{K_1} \setminus I_{K_2}) \cup (I_{K_1} \setminus (I_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap (V_{K_2} \cap O_{K_3})) \cup (V_{K_1} \cap (O_{K_2} \cap$
$O_{K_3})) \setminus O_{K_1})$
$= ((I_{K_1} \setminus I_{K_2}) \cup ((I_{K_1} \setminus I_{K_3}) \cap (I_{K_1} \setminus V_{K_1}))) \cup ((V_{K_1} \cap ((V_{K_2} \cap O_{K_3}) \cup (O_{K_2} \cap$
$O_{K_3}))) \setminus O_{K_1})$
$= ((I_{K_1} \setminus I_{K_2}) \cup ((I_{K_1} \setminus I_{K_3}) \cap I_{K_1})) \cup ((V_{K_1} \cap (O_{K_3} \cap (O_{K_2} \cup V_{K_2}))) \setminus O_{K_1})$
$= ((I_{K_1} \setminus I_{K_2}) \cup (I_{K_1} \setminus I_{K_3})) \cup ((V_{K_1} \cap O_{K_3}) \setminus O_{K_1})$
$= (I_{K_1} \setminus I_{K_3}) \cup ((V_{K_1} \setminus O_{K_1}) \cap O_{K_3})$
$= (I_{K_1} \setminus I_{K_3}) \cup (V_{K_1} \cap O_{K_3}) = I_{K'}$,

- $O_{K_2' \| K_3'} = (O_{K_2'} \setminus I_{K_3'}) \cup (O_{K_3'} \setminus I_{K_2'}) =$
$= (((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2})) \setminus ((I_{K_2} \setminus I_{K_3}) \cup (V_{K_2} \cap O_{K_3}))) \cup (((O_{K_2} \setminus O_{K_3}) \cup$
$(V_{K_2} \cap I_{K_3})) \setminus ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})))$
$= ((((O_{K_1} \setminus O_{K_2}) \setminus (I_{K_2} \setminus I_{K_3})) \cap ((O_{K_1} \setminus O_{K_2}) \setminus (V_{K_2} \cap O_{K_3}))) \cup (((V_{K_1} \cap I_{K_2}) \setminus$
$(I_{K_2} \setminus I_{K_3})) \cap ((V_{K_1} \cap I_{K_2}) \setminus (V_{K_2} \cap O_{K_3})))) \cup ((((O_{K_2} \setminus O_{K_3}) \setminus (I_{K_1} \setminus I_{K_2})) \cap ((O_{K_2} \setminus$
$O_{K_3}) \setminus (V_{K_1} \cap O_{K_2}))) \cup (((V_{K_2} \cap I_{K_3}) \setminus (I_{K_1} \setminus I_{K_2})) \cap ((V_{K_2} \cap I_{K_3}) \setminus (V_{K_1} \cap O_{K_2}))))$
$= (((((O_{K_1} \setminus O_{K_2}) \setminus I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{K_3})) \cap (((O_{K_1} \setminus O_{K_2}) \setminus V_{K_2}) \cup$
$((O_{K_1} \setminus O_{K_2}) \setminus O_{K_3}))) \cup ((V_{K_1} \cap (I_{K_2} \setminus (I_{K_2} \setminus I_{K_3}))) \cap (((V_{K_1} \cap I_{K_2}) \setminus V_{K_2}) \cup$
$((V_{K_1} \cap I_{K_2}) \setminus O_{K_3})))) \cup (((((O_{K_2} \setminus O_{K_3} \setminus I_{K_1}) \cup ((O_{K_2} \setminus O_{K_3}) \cap I_{K_2})) \cap (((O_{K_2} \setminus$
$O_{K_3}) \setminus V_{K_1}) \cup ((O_{K_2} \setminus O_{K_3}) \setminus O_{K_2}))) \cup ((I_{K_3} \cap (V_{K_2} \setminus (I_{K_1} \setminus I_{K_2}))) \cap (((V_{K_1} \cap$
$I_{K_3}) \setminus V_{K_1}) \cup ((V_{K_2} \cap I_{K_3}) \setminus O_{K_2}))))$
$= ((((O_{K_1} \setminus (O_{K_2} \cup I_{K_2})) \cup ((O_{K_1} \cap I_{K_3}) \setminus O_{K_2})) \cap ((O_{K_1} \setminus (O_{K_2} \cup V_{K_2})) \cup ((O_{K_1} \setminus$
$O_{K_2}) \setminus O_{K_3}))) \cup ((V_{K_1} \cap I_{K_2} \cap I_{K_3}) \cap ((V_{K_1} \cap (I_{K_2} \setminus V_{K_2})) \cup (V_{K_1} \cap (I_{K_2} \setminus O_{K_3}))))) \cup$
$((((O_{K_2} \setminus (O_{K_3} \cup I_{K_1})) \cup ((O_{K_2} \cap I_{K_2}) \setminus O_{K_3})) \cap ((O_{K_2} \setminus (O_{K_3} \cup V_{K_1})) \cup \emptyset)) \cup$
$((I_{K_3} \cap ((V_{K_2} \setminus I_{K_1}) \cup (V_{K_2} \cap I_{K_2}))) \cap ((V_{K_1} \setminus V_{K_1}) \cap I_{K_3}) \cup ((V_{K_2} \setminus O_{K_2}) \cap I_{K_3}))))$

$= (((((O_{K_1} \setminus O_{K_2}) \cap (O_{K_1} \setminus I_{K_2})) \cup \setminus) \cap (((O_{K_1} \setminus V_{K_2}) \setminus O_{K_2}) \cup (O_{K_1} \setminus (O_{K_2} \cup O_{K_3})))) \cup ((V_{K_1} \cap I_{K_2} \cap I_{K_3}) \cap ((V_{K_1} \cap I_{K_2}) \cup (V_{K_1} \cap I_{K_2})))) \cup (((((O_{K_2} \setminus O_{K_3}) \cap (O_{K_2} \setminus I_{K_1})) \cup \emptyset) \cap ((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1})) \cup ((I_{K_3} \cap (V_{K_2} \setminus I_{K_1})) \cap (\emptyset \cup (V_{K_2} \cap I_{K_3}))))$
$= (((((O_{K_1} \setminus O_{K_2}) \cap O_{K_1}) \cup \emptyset) \cap ((O_{K_1} \setminus O_{K_2}) \cup (O_{K_1} \setminus (O_{K_2} \cup O_{K_3})))) \cup ((V_{K_1} \cap I_{K_2} \cap I_{K_3}) \cap (V_{K_1} \cap I_{K_2}))) \cup ((((O_{K_2} \setminus O_{K_3}) \cap O_{K_2}) \cap ((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1})) \cup ((V_{K_2} \cap (I_{K_3} \setminus I_{K_1})) \cap (V_{K_2} \cap I_{K_3})))$
$= (((O_{K_1} \setminus O_{K_3}) \cap ((O_{K_1} \setminus O_{K_2}) \cup (O_{K_1} \setminus (O_{K_2} \cup O_{K_3})))) \cup ((V_{K_1} \cap I_{K_2} \cap I_{K_3}) \cap (V_{K_1} \cap I_{K_2}))) \cup (((O_{K_2} \setminus O_{K_3}) \cap ((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1})) \cup ((V_{K_2} \cap (I_{K_3} \setminus I_{K_1})) \cap (V_{K_2} \cap I_{K_3})))$
$= ((((O_{K_1} \setminus O_{K_2}) \cap (O_{K_1} \setminus O_{K_2})) \cup ((O_{K_1} \setminus O_{K_2}) \cap ((O_{K_1} \setminus O_{K_2}) \setminus O_{K_3}))) \cup (V_{K_1} \cap I_{K_2} \cap I_{K_3})) \cup (((O_{K_2} \setminus O_{K_3}) \cap ((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1})) \cup (((V_{K_2} \cap I_{K_3}) \setminus I_{K_1}) \cap (V_{K_2} \cap I_{K_3})))$
$= (((O_{K_1} \setminus O_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \setminus O_{K_3})) \cup (V_{K_1} \cap I_{K_2} \cap I_{K_3})) \cup (((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1}) \cup ((V_{K_2} \cap I_{K_3}) \setminus I_{K_1}))$
$= (((O_{K_1} \setminus O_{K_2}) \setminus (O_{K_3} \setminus (O_{K_1} \setminus O_{K_2}))) \cup (V_{K_1} \cap I_{K_2} \cap I_{K_3})) \cup (((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1}) \cup (V_{K_2} \cap (I_{K_3} \setminus I_{K_1})))$
$= ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2} \cap I_{K_3})) \cup (((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1}) \cup (V_{K_2} \cap (I_{K_3} \setminus I_{K_1})))$
$= (O_{K_1} \setminus O_{K_2}) \cup ((O_{K_2} \setminus O_{K_3}) \setminus V_{K_1}) \cup (V_{K_1} \cap I_{K_2} \cap I_{K_3}) \cup (V_{K_2} \cap (I_{K_3} \setminus I_{K_1}))$
$= ((O_{K_1} \cup ((O_{K_2} \setminus V_{K_1}) \setminus O_{K_3})) \setminus (O_{K_2} \setminus ((O_{K_2} \setminus V_{K_1}) \setminus O_{K_3}))) \cup (((V_{K_1} \cap I_{K_3}) \cap I_{K_2}) \cup ((V_{K_2} \cap I_{K_3}) \setminus I_{K_1}))$
$= (((O_{K_1} \cup (O_{K_2} \setminus V_{K_1})) \setminus (O_{K_3} \setminus O_{K_1})) \setminus ((O_{K_2} \cap O_{K_3}) \cup (O_{K_2} \setminus (O_{K_2} \setminus V_{K_1})))) \cup (((V_{K_1} \cap I_{K_3}) \cup ((V_{K_1} \cap I_{K_3}) \cap I_{K_2})) \setminus (I_{K_1} \setminus ((V_{K_1} \cap I_{K_3}) \cap I_{K_2})))$
$= ((((O_{K_1} \cup O_{K_2}) \setminus (V_{K_1} \setminus O_{K_1})) \setminus (O_{K_3} \setminus O_{K_1})) \setminus ((O_{K_2} \cap O_{K_3}) \cup (O_{K_2} \cap V_{K_1}))) \cup ((((V_{K_1} \cap I_{K_3}) \cup (V_{K_2} \cap I_{K_3})) \cap ((V_{K_2} \cap I_{K_3}) \cup I_{K_2}))) \setminus ((I_{K_1} \setminus (V_{K_1} \cap I_{K_3}) \cup (I_{K_1} \setminus I_{K_2})))$
$= ((((O_{K_1} \cup O_{K_2}) \setminus V_{K_1}) \setminus (O_{K_3} \setminus O_{K_1}) \setminus (O_{K_2} \cap (O_{K_3} \cup V_{K_1}))) \cup ((((V_{K_1} \cup V_{K_2}) \cap I_{K_3}) \cap ((V_{K_2} \cap I_{K_3}) \cup I_{K_2}))) \setminus (I_{K_1} \cup (I_{K_1} \setminus I_{K_2})))$
$= ((O_{K_1} \setminus (O_{K_3} \setminus O_{K_1})) \setminus (O_{K_2} \cap (O_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap I_{K_3}) \cap ((V_{K_2} \cap I_{K_3}) \cup I_{K_2})) \setminus (I_{K_1} \setminus (I_{K_2} \setminus I_{K_1})))$
$= (O_{K_1} \setminus (O_{K_2} \cap (O_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap I_{K_3} \cap V_{K_2}) \cup (V_{K_1} \cap I_{K_3} \cap I_{K_2})) \setminus I_{K_1})$
$= ((O_{K_1} \setminus O_{K_2}) \cup (O_{K_1} \setminus (O_{K_3} \cup V_{K_1}))) \cup (((V_{K_1} \cap (V_{K_2} \cap I_{K_3})) \cup (V_{K_1} \cap (I_{K_2} \cap I_{K_3})) \setminus I_{K_1})$
$= ((O_{K_1} \setminus O_{K_2}) \cup ((O_{K_1} \setminus O_{K_3}) \cap (O_{K_1} \setminus V_{K_1}))) \cup ((V_{K_1} \cap ((V_{K_2} \cap I_{K_3}) \cup (I_{K_2} \cap I_{K_3}))) \setminus I_{K_1})$
$= ((O_{K_1} \setminus O_{K_2}) \cup ((O_{K_1} \setminus O_{K_3}) \cap O_{K_1})) \cup ((V_{K_1} \cap (I_{K_3} \cap (I_{K_2} \cup V_{K_2}))) \setminus I_{K_1})$
$= ((O_{K_1} \setminus O_{K_2}) \cup (O_{K_1} \setminus O_{K_3})) \cup ((V_{K_1} \cap I_{K_3}) \setminus I_{K_1})$
$= (O_{K_1} \setminus O_{K_3}) \cup ((V_{K_1} \setminus I_{K_1}) \cap I_{K_3})$
$= (O_{K_1} \setminus O_{K_3}) \cup (V_{K_1} \cap I_{K_3}) = O_{K'}$ and

- $V_{K_2' \| K_3'} = V_{K_2'} \cup V_{K_3'} \cup (O_{K_2'} \cap I_{K_3'}) \cup (I_{K_2'} \cap O_{K_3'}) =$
  $= V_{K_2'} \cup V_{K_3'} \cup (((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2})) \cap ((I_{K_2} \setminus I_{K_3}) \cup (V_{K_2} \cap O_{K_3}))) \cup$

$$(((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})) \cap ((O_{K_2} \setminus O_{K_3}) \cup (V_{K_2} \cap I_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (((O_{K_1} \setminus O_{K_2}) \cap (I_{K_2} \setminus I_{K_3})) \cup ((V_{K_1} \cap I_{K_2}) \cap (I_{K_2} \setminus I_{K_3})) \cup ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_2} \cap O_{K_3})) \cup ((V_{K_1} \cap I_{K_2}) \cap (V_{K_2} \cap O_{K_3}))) \cup (((I_{K_1} \setminus I_{K_2}) \cap (O_{K_2} \setminus O_{K_3})) \cup ((V_{K_1} \cap O_{K_2}) \cap (O_{K_2} \setminus O_{K_3})) \cup ((I_{K_1} \setminus I_{K_2}) \cap V_{K_2} \cap I_{K_3})) \cup ((V_{K_1} \cap O_{K_2}) \cap (V_{K_2} \cap I_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (\emptyset \cup (((V_{K_1} \cap I_{K_2}) \cap I_{K_2}) \setminus I_{K_3}) \cup ((O_{K_1} \cap V_{K_2} \cap O_{K_3}) \setminus O_{K_2}) \cup \emptyset) \cup (\emptyset \cup (((V_{K_1} \cap O_{K_2}) \cap O_{K_2}) \setminus O_{K_3}) \cup ((V_{K_2} \cap I_{K_1} \cap I_{K_3}) \setminus I_{K_2}) \cup \emptyset)$$

$$= V_{K_2'} \cup V_{K_3'} \cup (((V_{K_1} \cap I_{K_2}) \setminus I_{K_3}) \cup \emptyset) \cup (((V_{K_1} \cap O_{K_2}) \setminus O_{K_3}) \cup \emptyset)$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap (I_{K_2} \setminus I_{K_3})) \cup (V_{K_1} \cap (O_{K_2} \setminus O_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap ((I_{K_2} \setminus I_{K_3}) \cup (O_{K_2} \setminus O_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap ((I_{K_2} \cup (O_{K_2} \setminus O_{K_3})) \setminus (I_{K_3} \setminus (O_{K_2} \setminus O_{K_3}))))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap ((I_{K_2} \cup O_{K_2}) \setminus (O_{K_3}) \setminus I_{K_2})) \setminus I_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap (((I_{K_2} \cup O_{K_2}) \setminus O_{K_3}) \setminus I_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap ((I_{K_2} \cup O_{K_2}) \setminus (I_{K_3} \cup O_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap ((E_{K_2} \setminus V_{K_2}) \setminus (E_{K_3} \setminus V_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap (E_{K_2} \setminus V_{K_2})) \setminus (E_{K_3} \setminus V_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (((V_{K_1} \cap E_{K_2}) \setminus V_{K_2}) \setminus (E_{K_3} \setminus V_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap E_{K_2}) \setminus (V_{K_2} \cup (E_{K_3} \setminus V_{K_3})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap E_{K_2}) \setminus ((V_{K_2} \cup E_{K_3}) \setminus (V_{K_3} \setminus V_{K_2})))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap E_{K_2}) \setminus ((V_{K_2} \cup E_{K_3}) \setminus \emptyset))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap E_{K_2}) \cap (V_{K_2} \cup E_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (((V_{K_1} \cap E_{K_2}) \setminus V_{K_2}) \cap ((V_{K_1} \cap E_{K_2}) \setminus E_{K_3}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \cap (E_{K_2} \setminus V_{K_2}) \cap ((V_{K_1} \setminus E_{K_3}) \cap E_{K_2}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \cap (V_{K_1} \setminus E_{K_3}) \cap E_{K_2} \cap (E_{K_2} \setminus V_{K_2}))$$

$$= V_{K_2'} \cup V_{K_3'} \cup ((V_{K_1} \setminus E_{K_3}) \cap (E_{K_2} \setminus V_{K_2}))$$

$$= (V_{K_2'} \cup V_{K_3'} \cup (V_{K_1} \setminus E_{K_3})) \cap (V_{K_2'} \cup V_{K_3'} \cup (E_{K_2} \setminus V_{K_2}))$$

$$= ((V_{K_1} \setminus E_{K_2}) \cup (V_{K_2} \setminus E_{K_3}) \cup (V_{K_1} \setminus E_{K_3})) \cap ((V_{K_1} \setminus E_{K_2}) \cup (E_{K_2} \setminus V_{K_2}) \cup (V_{K_2} \setminus E_{K_3}))$$

$$= ((V_{K_1} \setminus (E_{K_2} \cap E_{K_3})) \cup (V_{K_2} \setminus E_{K_3})) \cap (((V_{K_1} \cup (E_{K_2} \setminus V_{K_2})) \setminus (E_{K_2} \setminus (E_{K_2} \setminus V_{K_2})) \cup (V_{K_2} \setminus E_{K_3}))$$

$$= ((V_{K_1} \setminus E_{K_3}) \cup (V_{K_2} \setminus E_{K_3})) \cap (((V_{K_1} \cup (E_{K_2} \setminus V_{K_2})) \setminus (E_{K_2} \cap V_{K_2})) \cup (V_{K_2} \setminus E_{K_3}))$$

$$= ((V_{K_1} \cup V_{K_2}) \setminus E_{K_3}) \cap (((V_{K_1} \cup (E_{K_2} \setminus V_{K_2})) \setminus V_{K_2}) \cup (V_{K_2} \setminus E_{K_3}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap ((V_{K_1} \cup (E_{K_2} \setminus V_{K_2}) \cup (V_{K_2} \setminus E_{K_3})) \setminus (V_{K_2} \setminus (V_{K_2} \setminus E_{K_3})))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap ((V_{K_1} \cup (((E_{K_2} \setminus V_{K_2}) \cup V_{K_2}) \setminus (E_{K_3} \setminus (E_{K_2} \setminus V_{K_2})))) \setminus (V_{K_2} \cap E_{K_3}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap (V_{K_1} \cup (E_{K_2} \setminus ((E_{K_3} \setminus E_{K_2}) \cup (E_{K_3} \cap V_{K_2})))) \setminus (V_{K_2} \cap E_{K_3}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap ((V_{K_1} \cup (E_{K_2} \setminus (\emptyset \cup (V_{K_2} \cap E_{K_3})))) \setminus (V_{K_2} \cap E_{K_3}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap ((V_{K_1} \cup (E_{K_2} \setminus (E_{K_3} \cap V_{K_2}))) \setminus (E_{K_3} \cap V_{K_2}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap (((E_{K_2} \cup V_{K_1}) \setminus ((E_{K_3} \cap V_{K_2}) \setminus V_{K_1}))) \setminus (E_{K_3} \cap V_{K_2}))$$

$$= (V_{K_1} \setminus E_{K_3}) \cap ((E_{K_2} \cup V_{K_1}) \setminus (((E_{K_3} \cap V_{K_2}) \setminus V_{K_1}) \cup (E_{K_3} \cap V_{K_2})))$$

$= (V_{K_1} \setminus E_{K_3}) \cap ((E_{K_2} \cup V_{K_1}) \setminus (E_{K_3} \cap V_{K_2}))$
$= ((V_{K_1} \cup E_{K_2}) \cap (V_{K_1} \setminus E_{K_3})) \setminus (E_{K_3} \setminus V_{K_2}) = ((V_{K_1} \cap (V_{K_1} \cup E_{K_2})) \setminus (E_{K_3} \setminus (V_{K_1} \cup E_{K_2}))) \setminus (E_{K_3} \setminus V_{K_2}) = (V_{K_1} \setminus ((E_{K_3} \setminus V_{K_1}) \cap (E_{K_3} \setminus E_{K_2}))) \setminus (E_{K_3} \setminus V_{K_2}) = V_{K_1} \setminus (E_{K_3} \setminus V_{K_2}) = (V_{K_1} \setminus E_{K_3}) \setminus (V_{K_2} \setminus V_{K_1}) = V_{K_1} \setminus E_{K_3} = V_{K'}$.

□

B.3 Proof of Theorem 6.3

Theorem 6.3. *Let K_1 and K_2 be two components and Env an environment compatible with both K_1 and K_2 such that $Env = Env_1 \parallel Env_2$. Then $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1$.*

Proof. $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel (Env_1 \parallel Env_2) \parallel Env' \preceq K_2 \parallel (Env_1 \parallel Env_2) \parallel K' \parallel Env'$
$K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1 \Leftrightarrow (K_1 \parallel Env_1) \parallel Env_2 \parallel Env'' \preceq (K_2 \parallel Env_1) \parallel Env_2 \parallel K'' \parallel Env''$

The two relations identical based on the associativity of $\parallel$, where

1. $Env' = Env'' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})), (I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})), \emptyset, \emptyset, D_{Env'}, 2_0^{[\mathbb{R}+]})$
   - $I_{Env'} = O_{K_1} \setminus I_{Env_1 \parallel Env_2} =$
     $= O_{K_1} \setminus ((I_{Env_1} \setminus O_{Env_1}) \cup (I_{Env_2} \setminus O_{Env_1}))$
     $= (O_{K_1} \setminus (I_{Env_1} \setminus O_{Env_2})) \cap (O_{K_1} \setminus (I_{Env_2} \setminus O_{Env_1}))$
     $= ((O_{K_1} \cap O_{Env_2}) \cup (O_{K_1} \setminus I_{Env_1})) \cap ((O_{K_1} \cap O_{Env_1}) \cup (O_{K_1} \setminus I_{Env_2}))$
     $= (\emptyset \cup (O_{K_1} \setminus I_{Env_1})) \cap (\emptyset \cup (O_{K_1} \setminus I_{Env_2}))$
     $= (O_{K_1} \setminus I_{Env_1}) \cap (O_{K_1} \setminus I_{Env_2})$
     $= O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})$

     $I_{Env''} = O_{K_1 \parallel Env_1} \setminus I_{Env_2} =$
     $= ((O_{K_1} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_1})) \setminus I_{Env_2}$
     $= ((O_{K_1} \setminus I_{Env_1}) \setminus I_{Env_2}) \cup ((O_{Env_1} \setminus I_{K_1}) \setminus I_{Env_2})$
     $= (O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})) \cup (O_{Env_1} \setminus (I_{K_1} \cup I_{Env_2}))$
     $= (O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})) \cup \emptyset$
     $= O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})$
   - $O_{Env'} = I_{K_1} \setminus O_{Env_1 \parallel Env_2} =$
     $= I_{K_1} \setminus ((O_{Env_1} \setminus I_{Env_2}) \cup (O_{Env_2} \setminus I_{Env_1}))$
     $= (I_{K_1} \setminus (O_{Env_1} \setminus I_{Env_2})) \cap (I_{K_1} \setminus (O_{Env_2} \setminus I_{Env_1}))$
     $= ((I_{K_1} \cap I_{Env_2}) \cup (I_{K_1} \setminus O_{Env_1})) \cap ((I_{K_1} \cap I_{Env_1}) \cup (I_{K_1} \setminus O_{Env_2}))$

$= (\emptyset \cup (I_{K_1} \setminus O_{Env_1})) \cap (\emptyset \cup (I_{K_1} \setminus O_{Env_2}))$
$= (I_{K_1} \setminus O_{Env_1}) \cap (I_{K_1} \setminus O_{Env_2})$
$= I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})$

$O_{Env''} = I_{K_1 \| Env_1} \setminus O_{Env_2} =$
$= ((I_{K_1} \setminus O_{Env_1}) \setminus O_{Env_2}) \cup ((I_{Env_1} \setminus O_{K_1}) \setminus O_{Env_2})$
$= (I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})) \cup (I_{Env_1} \setminus (O_{K_1} \cup O_{Env_2}))$
$= (I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})) \cup \emptyset$
$= I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})$

- $V_{Env'} = \emptyset = V_{Env''}$

2. $K' = K'' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_1})), (V_{K_1} \setminus E_{K_2}), \emptyset, D_{K'}, 2_0^{[\mathbb{R}_+]})$

   - $I_{K'} = (I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})$

     $I_{K''} = (I_{K_1 \| Env_1} \setminus I_{K_2 \| Env_1}) \cup (V_{K_1 \| Env_1} \cap O_{K_2 \| Env_1}) =$
     $= (((I_{K_1} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_1})) \setminus ((I_{K_2} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_2}))) \cup$
     $((V_{K_1} \cup V_{Env_1} \cup (O_{K_1} \cap I_{Env_1}) \cup (I_{K_1} \cap O_{Env_1})) \cap O_{K_2 \| Env_1})$
     $= (((I_{K_1} \setminus O_{Env_1}) \setminus (I_{K_2} \setminus O_{Env_1})) \cap ((I_{K_1} \setminus O_{Env_1}) \setminus (I_{Env_1} \setminus O_{K_2}))) \cup (((I_{Env_1} \setminus$
     $O_{K_1}) \setminus (I_{K_2} \setminus O_{Env_1})) \cap ((I_{Env_1} \setminus O_{K_1}) \setminus (I_{Env_1} \setminus O_{K_2}))) \cup (V_{K_1} \cap O_{K_2 \| Env_1}) \cup$
     $(V_{Env_1} \cap O_{K_2 \| Env_1}) \cup ((O_{K_1} \cap I_{Env_1}) \cap O_{K_2 \| Env_1}) \cup ((I_{K_1} \cap O_{Env_1}) \cap O_{K_2 \| Env_1}$
     $= ((I_{K_1} \setminus (O_{Env_1} \cup (I_{K_2} \setminus O_{Env_1}))) \cap (I_{K_1} \setminus (O_{Env_1} \cup (I_{Env_1} \setminus O_{K_2})))) \cup$
     $((I_{Env_1} \setminus (O_{K_2} \cup (I_{K_2} \setminus O_{Env_1}))) \cap (I_{Env_1} \setminus (O_{K_1} \cup (I_{Env_1} \setminus O_{K_2})))) \cup (V_{K_1} \cap$
     $((O_{K_2} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_2}))) \cup (V_{Env_1} \cap ((O_{K_2} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_2}))) \cup$
     $((O_{K_1} \cap I_{Env_1} \cap ((O_{K_2} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_2}))) \cup ((O_{Env_1} \cap I_{K_1}) \cap ((O_{K_2} \setminus$
     $I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_2})))$
     $= ((I_{K_1} \setminus ((O_{Env_1} \cup I_{K_2}) \setminus (O_{Env_1} \setminus O_{Env_1}))) \cap (I_{K_1} \setminus ((O_{Env_1} \cup I_{Env_1}) \setminus$
     $(O_{K_2} \setminus O_{Env_1})))) \cup ((I_{Env_1} \setminus ((O_{K_2} \cup I_{K_2}) \setminus (O_{Env_1} \setminus O_{K_2}))) \cap (I_{Env_1} \setminus$
     $((O_{K_1} \cup I_{Env_1}) \setminus (O_{K_2} \setminus O_{K_1})))) \cup (V_{K_1} \cap (O_{K_2} \setminus I_{Env_1})) \cup (V_{K_1} \cap (O_{Env_1} \setminus$
     $I_{K_2})) \cup \emptyset \cup ((O_{K_1} \cap I_{Env_1}) \cap (O_{K_2} \setminus I_{Env_1})) \cup ((O_{K_1} \cap I_{Env_1}) \cap (O_{Env_1} \setminus$
     $I_{K_2})) \cup ((O_{Env_1} \cap I_{K_1}) \cap (O_{K_2} \setminus I_{Env_1})) \cup ((O_{Env_1} \cap I_{K_1}) \cap (O_{Env_1} \setminus I_{K_1}))$
     $= ((I_{K_1} \setminus (O_{Env_1} \cup I_{K_2})) \cap (I_{K_1} \setminus ((O_{Env_1} \cup I_{Env_1}) \setminus O_{K_2}))) \cup ((I_{Env_1} \setminus$
     $((O_{K_2} \cup I_{K_2}) \setminus O_{Env_1})) \cap (I_{Env_1} \setminus ((O_{K_1} \cup I_{Env_1}) \setminus (O_{K_2} \setminus O_{K_1})))) \cup (V_{K_1} \cap$
     $O_{K_2}) \cup \emptyset \cup ((O_{K_1} \cap I_{Env_1} \cap O_{K_2}) \setminus I_{Env_1}) \cup ((O_{K_1} \cap I_{Env_1} \cap O_{Env_1}) \setminus I_{K_2}) \cup$
     $((O_{Env_1} \cap I_{K_1} \cap O_{K_2}) \setminus I_{K_2}) \cup ((O_{Env_1} \cap I_{K_1} \cap O_{Env_1}) \setminus I_{K_2}$
     $= ((I_{K_1} \setminus (O_{Env_1} \cup I_{K_2})) \cap ((I_{K_1} \cap O_{K_2}) \cup (I_{K_1} \setminus (O_{Env_1} \cup I_{Env_1})))) \cup$
     $(((I_{Env_1} \cap O_{Env_1}) \cup (I_{Env_1} \setminus (O_{K_2} \cup I_{K_2}))) \cap ((I_{Env_1} \cap ((O_{K_2} \setminus O_{K_1})) \cup$
     $(I_{Env_1} \setminus (O_{K_1} \cup I_{Env_1})))) \cup (V_{K_1} \cap O_{K_2}) \cup \emptyset \cup \emptyset \cup \emptyset \cup ((O_{Env_1} \cap I_{K_1}) \setminus I_{K_2})$
     $= ((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cap (\emptyset \cup ((I_{K_1} \setminus O_{Env_1}) \cap (I_{K_1} \setminus I_{Env_1})))) \cup ((\emptyset \cup$
     $((I_{Env_1} \setminus O_{K_2}) \cap (I_{Env_1} \setminus I_{K_2}))) \cap ((I_{Env_1} \cap (O_{K_2} \setminus O_{K_1})) \cup ((I_{Env_1} \setminus O_{K_1}) \cap$
     $(I_{Env_1} \setminus I_{Env_1})))) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
     $= (((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cap ((I_{K_1} \setminus O_{Env_1}) \cap I_{K_1})) \cup (((I_{Env_1} \setminus O_{K_2}) \cap I_{Env_1}) \cap$

$((I_{Env_1} \cap (O_{K_2} \setminus O_{K_1})) \cup ((I_{Env_1} \setminus O_{K_1}) \cap \emptyset))) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= (((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cap (I_{K_1} \setminus O_{Env_1})) \cup ((I_{Env_1} \setminus O_{K_2}) \cap (I_{Env_1} \cap (O_{K_2} \setminus O_{K_1}))) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= ((((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cap I_{K_1}) \setminus O_{Env_1}) \cup ((I_{Env_1} \cap (O_{K_2} \setminus O_{K_1}) \cap I_{Env_1}) \setminus O_{K_2}) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= ((((I_{K_1} \setminus I_{K_2}) \cap I_{K_1}) \setminus O_{Env_1}) \setminus O_{Env_1}) \cup ((I_{Env_1} \cap (O_{K_2} \setminus O_{K_1})) \setminus O_{K_2}) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= ((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cup (I_{Env_1} \cap ((O_{K_2} \setminus O_{K_1}) \setminus O_{K_2}))) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= ((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cup (I_{Env_1} \cap \emptyset) \cup (V_{K_1} \cap O_{K_2}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1})$
$= ((I_{K_1} \setminus I_{K_2}) \setminus O_{Env_1}) \cup ((I_{K_1} \setminus I_{K_2}) \cap O_{Env_1}) \cup (V_{K_1} \cap O_{K_2}) = (I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})$

- $O_{K'} = (O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_1})$

  $O_{K''} = (O_{K_1\|Env_1} \setminus O_{K_2\|Env_1}) \cup (V_{K_1\|Env_1} \cap I_{K_2\|Env_1} =$
  $= (((O_{K_1} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_1})) \setminus ((O_{K_2} \setminus I_{Env_1}) \cup (O_{Env_1} \setminus I_{K_2}))) \cup ((V_{K_1} \cup V_{Env_1} \cup (I_{K_1} \cap O_{Env_1}) \cup (O_{K_1} \cap I_{Env_1})) \cap I_{K_2\|Env_1})$
  $= (((O_{K_1} \setminus I_{Env_1}) \setminus (O_{K_2} \setminus I_{Env_1})) \cap ((O_{K_1} \setminus I_{Env_1}) \setminus (O_{Env_1} \setminus I_{K_2}))) \cup (((O_{Env_1} \setminus I_{K_1}) \setminus (O_{K_2} \setminus I_{Env_1})) \cap ((O_{Env_1} \setminus I_{K_1}) \setminus (O_{Env_1} \setminus I_{K_2}))) \cup (V_{K_1} \cap I_{K_2\|Env_1}) \cup (V_{Env_1} \cap I_{K_2\|Env_1}) \cup ((I_{K_1} \cap O_{Env_1}) \cap I_{K_2\|Env_1}) \cup ((O_{K_1} \cap I_{Env_1}) \cap I_{K_2\|Env_1}$
  $= ((O_{K_1} \setminus (I_{Env_1} \cup (O_{K_2} \setminus I_{Env_1}))) \cap (O_{K_1} \setminus (I_{Env_1} \cup (O_{Env_1} \setminus I_{K_2})))) \cup ((O_{Env_1} \setminus (I_{K_2} \cup (O_{K_2} \setminus I_{Env_1}))) \cap (O_{Env_1} \setminus (I_{K_1} \cup (O_{Env_1} \setminus I_{K_2})))) \cup (V_{K_1} \cap ((I_{K_2} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_2}))) \cup (V_{Env_1} \cap ((I_{K_2} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_2}))) \cup ((I_{K_1} \cap O_{Env_1} \cap ((I_{K_2} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_2}))) \cup ((I_{Env_1} \cap O_{K_1}) \cap ((I_{K_2} \setminus O_{Env_1}) \cup (I_{Env_1} \setminus O_{K_2})))$
  $= ((O_{K_1} \setminus ((I_{Env_1} \cup O_{K_2}) \setminus (I_{Env_1} \setminus I_{Env_1}))) \cap (O_{K_1} \setminus ((I_{Env_1} \cup O_{Env_1}) \setminus (I_{K_2} \setminus I_{Env_1})))) \cup ((O_{Env_1} \setminus ((I_{K_2} \cup O_{K_2}) \setminus (I_{Env_1} \setminus I_{K_2}))) \cap (O_{Env_1} \setminus ((I_{K_1} \cup O_{Env_1}) \setminus (I_{K_2} \setminus I_{K_1})))) \cup (V_{K_1} \cap (I_{K_2} \setminus O_{Env_1})) \cup (V_{K_1} \cap (I_{Env_1} \setminus O_{K_2})) \cup \emptyset \cup ((I_{K_1} \cap O_{Env_1}) \cap (I_{K_2} \setminus O_{Env_1})) \cup ((I_{K_1} \cap O_{Env_1}) \cap (I_{Env_1} \setminus O_{K_2})) \cup ((I_{Env_1} \cap O_{K_1}) \cap (I_{K_2} \setminus O_{Env_1})) \cup ((I_{Env_1} \cap O_{K_1}) \cap (I_{Env_1} \setminus O_{K_1}))$
  $= ((O_{K_1} \setminus (I_{Env_1} \cup O_{K_2})) \cap (O_{K_1} \setminus ((I_{Env_1} \cup O_{Env_1}) \setminus I_{K_2}))) \cup ((O_{Env_1} \setminus ((I_{K_2} \cup O_{K_2}) \setminus I_{Env_1})) \cap (O_{Env_1} \setminus ((I_{K_1} \cup O_{Env_1}) \setminus (I_{K_2} \setminus I_{K_1})))) \cup (V_{K_1} \cap I_{K_2}) \cup \emptyset \cup ((I_{K_1} \cap O_{Env_1} \cap I_{K_2}) \setminus O_{Env_1}) \cup ((I_{K_1} \cap O_{Env_1} \cap I_{Env_1}) \setminus O_{K_2}) \cup ((I_{Env_1} \cap O_{K_1} \cap I_{K_2}) \setminus O_{K_2}) \cup ((I_{Env_1} \cap O_{K_1} \cap I_{Env_1}) \setminus O_{K_2}$
  $= ((O_{K_1} \setminus (I_{Env_1} \cup O_{K_2})) \cap ((O_{K_1} \cap I_{K_2}) \cup (O_{K_1} \setminus (I_{Env_1} \cup O_{Env_1})))) \cup (((O_{Env_1} \cap I_{Env_1}) \cup (O_{Env_1} \setminus (I_{K_2} \cup O_{K_2}))) \cap ((O_{Env_1} \cap ((I_{K_2} \setminus I_{K_1})) \cup (O_{Env_1} \setminus (I_{K_1} \cup O_{Env_1})))) \cup (V_{K_1} \cap I_{K_2}) \cup \emptyset \cup \emptyset \cup \emptyset \cup ((I_{Env_1} \cap O_{K_1}) \setminus O_{K_2})$
  $= ((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cap (\emptyset \cup ((O_{K_1} \setminus I_{Env_1}) \cap (O_{K_1} \setminus O_{Env_1})))) \cup ((\emptyset \cup$

$((O_{Env_1} \setminus I_{K_2}) \cap (O_{Env_1} \setminus O_{K_2}))) \cap ((O_{Env_1} \cap (I_{K_2} \setminus I_{K_1})) \cup ((O_{Env_1} \setminus I_{K_1}) \cap$ $(O_{Env_1} \setminus O_{Env_1})))) \cup (V_{K_1} \cap I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= (((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cap ((O_{K_1} \setminus I_{Env_1}) \cap O_{K_1})) \cup (((O_{Env_1} \setminus I_{K_2}) \cap$ $O_{Env_1}) \cap ((O_{Env_1} \cap (I_{K_2} \setminus I_{K_1})) \cup ((O_{Env_1} \setminus I_{K_1}) \cap \emptyset))) \cup (V_{K_1} \cap I_{K_2}) \cup$ $((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= (((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cap (O_{K_1} \setminus I_{Env_1})) \cup ((O_{Env_1} \setminus I_{K_2}) \cap (O_{Env_1} \cap$ $(I_{K_2} \setminus I_{K_1}))) \cup (V_{K_1} \cap I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= ((((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cap O_{K_1}) \setminus I_{Env_1}) \cup ((O_{Env_1} \cap (I_{K_2} \setminus I_{K_1}) \cap O_{Env_1}) \setminus$ $I_{K_2}) \cup (V_{K_1} \cap I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= ((((O_{K_1} \setminus O_{K_2}) \cap O_{K_1}) \setminus I_{Env_1}) \setminus I_{Env_1}) \cup ((O_{Env_1} \cap (I_{K_2} \setminus I_{K_1})) \setminus I_{K_2}) \cup$ $(V_{K_1} \cap I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= ((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cup (O_{Env_1} \cap ((I_{K_2} \setminus I_{K_1}) \setminus I_{K_2}))) \cup (V_{K_1} \cap I_{K_2}) \cup$ $((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= ((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cup (O_{Env_1} \cap \emptyset) \cup (V_{K_1} \cap I_{K_2}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1})$

$= ((O_{K_1} \setminus O_{K_2}) \setminus I_{Env_1}) \cup ((O_{K_1} \setminus O_{K_2}) \cap I_{Env_1}) \cup (V_{K_1} \cap I_{K_2}) = (O_{K_1} \setminus$ $O_{K_2}) \cup (V_{K_1} \cap I_{K_2})$

- $V_{K'} = V_{K_1} \setminus E_{K_2}$

  $V_{K''} = V_{K_1 \| Env_1} \setminus E_{K_2 \| Env_1} =$

  $= (V_{K_1} \cup V_{Env_1} \cup (I_{K_1} \cap O_{Env_1}) \cup (I_{Env_1} \cap O_{K_1})) \setminus (E_{K_2} \cup E_{Env_1})$

  $= (V_{K_1} \setminus (E_{K_2} \cup E_{Env_1})) \cup (V_{Env_1} \setminus (E_{K_2} \cup E_{Env_1})) \cup ((I_{K_1} \cap O_{Env_1}) \setminus$ $(E_{K_2} \cup E_{Env_1})) \cup ((O_{K_1} \cap I_{Env_1}) \setminus (E_{K_2} \cup E_{Env_1}))$

  $= ((V_{K_1} \setminus E_{K_2}) \setminus E_{Env_1}) \cup \emptyset \cup \emptyset \cup \emptyset$

  $= V_{K_1} \setminus E_{K_2}$

□