42, A Component-Based Approach to Virtual Prototyping of Heterogeneous Embedded Systems

Tayeb Bouhadiba

UNIVERSITE DE GRENOBLE

No° attribué par la bibliothèque

THESE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE DE GRENOBLE

Spécialité : Informatique

préparée au Laboratoire VERIMAG

dans le cadre de l'École Doctorale Mathématiques,
Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement

par

Tayeb Sofiane BOUHADIBA

le 15 Septembre 2010

42, Une Approche à Composants pour le Prototypage Virtuel des Systèmes Embarqués Hétérogènes

42, A Component-Based Approach to Virtual Prototyping of Heterogeneous Embedded Systems

---

Directrice de thèse : Florence MARANINCHI

JURY:

Marc POUZET	Pr.	ENS Paris	Rapporteur et Président
Lionel SEINTURIER	Pr.	Université de Lille	Rapporteur
Jean-Bernard STEFANI	D.R	INRIA Rhône-Alpes	Examinateur
Florence MARANINCHI	Pr.	Grenoble INP	Directrice de thèse

Remerciements

Les travaux qui sont présentés dans cette thèse n'auraient pu être possible sans l'aide de plusieurs personnes. Ne pouvant citer tout le monde, je tiens à remercier tous ceux qui ont contribué, de près ou de loin à l'aboutissement de cette thèse.

Je tiens en particulier à remercier ma directrice de thèse Florence Maraninchi pour son encadrement, ses précieux conseils, pour tout ce que j'ai pu apprendre au travers des 42^{∞} discussions et pour m'avoir permis de développer mes idées.

C'est un honneur pour moi que Marc Pouzet, Lionel Seinturier et Jean-Bernard Stefani aient accepté de faire partie de mon jury de thèse. Un grand merci à eux pour avoir sacrifié un moment précieux de leur temps afin de lire et juger ma thèse.

Je remercie aussi les membres de l'équipe Synchrone du laboratoire Verimag. Leur disponibilité pour répondre à mes multiples questions m'ont beaucoup aidé à approfondir ma culture dans divers domaines. Une mention spéciale pour les membres du Bureau N°9, les anciens et les nouveaux.

Je reste reconnaissant à tout ceux qui ont pris le temps de relire (un bout de) ma thèse pour me rapporter des erreurs ou des correctifs. 42milles merci à Florence, David, Stéphane, Giovanni, Thomas, Kevin, Matthieu, Nicolas, Julien, Selma, Sophie, Laurie et tous ceux que j'ai pu oublié.

Il ne faut pas oublier l'ensemble du personnel administratif, merci à toute l'équipe qui entoure Christine pour leur travail remarquable. Aussi, un grand merci pour notre administrateur système Jean-Nöel (alias **root**) qui veille à maintenir le système au point, et qui m'a souvent rattrapé le paralysant *rm -rf **.

Je remercie encore les membres du laboratoire Verimag avec qui j'ai eu le plaisir de partager beaucoup d'activités extra-scientifiques, que ce soit pour prendre un café, faire ski, jouer au foot,...

Je ne peux pas oublier mes amis et cousins Ahmed, Amine(s), Chicho, Farah, Farid, Hichem, Kader, Khaled, Tamtam,... pour le temps d'un petit coup de fil, d'une discussion, d'une escapade ou d'une partie de PES. Les moments que j'ai partagé avec eux sont inoubliables et m'ont beaucoup aidé a tenir le coup.

Enfin, je pense que je ne serais pas arrivé là sans l'aide précieuse de ma famille côté Bouhadiba, Boukbir et Benbernou. Un grand merci à mes parents pour m'avoir toujours soutenu durant mes études et m'avoir offert la possibilité de continuer et d'aller aussi loin. Merci à vous Sosso, Radia et Imène, les coups de fil et visites de mes sœurs m'ont toujours apporté un énorme réconfort.

Enfin, je terminerai mes remerciements par celle qui partage ma vie. Merci à toi Ismahène pour le soutien que tu m'a apporté durant cette thèse et pour avoir partagé les moments difficiles par lesquels j'ai pu passé.

Contents

Chapter 1'

Introduction (In French)

1'.1 Les Systèmes Embarqués et leur Cycle de Développement

Les systèmes embarqués sont des systèmes informatiques omniprésents dans notre vie quotidienne. Les domaines où ils sont utilisés varient des systèmes critiques (avionique, centrales nucléaires, transport, etc.) à l'électronique grand publique (appareils photo, smartphones, etc.).

Comparés aux ordinateurs personnels, la particularité des systèmes embarqués réside dans leur architecture matérielle. En effet, l'architecture matérielle d'un système embarqué est souvent dédiée à celui-ci, et change d'une application à une autre. Les choix de conception du matériel sont guidés par la fonctionnalité du système, ainsi que par des propriétés non-fonctionnelles telle que la consommation en énergie.

Dans un système embarqué, les parties matérielles et logicielles sont fortement couplées. De ce fait, il est indispensable de les concevoir ensemble. Cependant, trouver une solution optimale pour de tels systèmes est difficile à cause des différents paramètres qui doivent être pris en compte (vitesse de calcul, consommation d'énergie, taille de la mémoire, surface disponible, etc.). Un autre paramètre à prendre en compte est la fabrication de la partie matérielle, qui est souvent disponible très tard dans le cycle de développement. En conséquence, la conception des système embarqués repose sur des techniques de *prototypage virtuel*.

Un prototype virtuel est un modèle exécutable d'un système. La modélisation nous permet d'étudier un système, très tôt, avant que le système ne soit disponible. En plus : 1) si le langage de spécification utilisé pour écrire les modèles est exécutable, nous pouvons observer le comportement du système réel en faisant des simulations ; 2) si les modèles sont formellement définis, nous avons la possibilité de vérifier quelques propriétés du système en appliquant les méthodes de *validation formelle* ; 3) le système final peut être généré automatiquement depuis son modèle ; cette technique est souvent appelée *développement (conception, etc.) dirigés par les modèles*.

Ecrire des modèles pour systèmes embarqués est parfois difficile. Une des causes de cette difficulté est *l'hétérogénéité des systèmes embarqués* : Ils sont composés de matériel et de logiciel ; le matériel peut contenir des composants analogiques et numériques ; les composants d'un système sont extrêmement concurrents, le modèle de concurrence peut varier du pure synchrone au pure asynchrone ; etc. En plus de l'hétérogénéité inhérente aux systèmes embarqués eux-même, une autre forme d'hétérogénéité apparaît dans le flot de conception : la conception de ces systèmes implique une expertise dans plusieurs domaines de l'ingénierie incluant l'électronique, l'automatique, l'informatique, etc. Dans chaque domaine, il existe une multitude d'outils et de formalismes pour répondre aux questions cruciales relatives à ce domaine. Chaque formalisme ou outil a ses propres notions pour modéliser les aspects relatifs à la concurrence, et le temps.

Pour palier à l'hétérogénéité des systèmes embarqués, un environnement de modélisation doit permettre la modélisation des composants hétérogènes du système. De plus, il doit permettre de décrire différents types de concurrence, de communication, de temps, de synchronisation, etc. Ces notions, sont définies par la notion de *MoCC* (*Model of Computation and Communication*).

Une autre notion relative aux travaux effectués dans cette thèse est celle des *composants* pour les systèmes embarqués. Les approches basées sur les composants ont fait leur apparition en réponse à la complexité croissante des systèmes embarqués et aux contraintes de temps de mise sur la marché. Dans l'industrie électronique (la partie matérielle), la notion de composants existe depuis bien longtemps. En effet, les *IPs (Intellectual Properties)* sont des composants électroniques prêts à l'usage. Ils sont vendus sous forme de composants physiques (prêts à être intégrés dans une plateforme matérielle), ou sous forme de spécifications synthétisables (doivent être intégrées durant le flot de conception). La notion de composant pour le matériel est assez générique grâce à l'adoption du *MoCC* universel pour les circuits synchrone.

Dans l'industrie du logiciel, les approches à composants sont nombreuses. Cependant, la notion de composant pour le logiciel n'est pas aussi générique que celle des composants pour le matériel. Ceci est dû à la multitude de *MoCCs* où ils peuvent être utilisés (threads, processus, programmes, programmation par événements, etc.)

1'.2 Le Prototypage par Composants et ses Challenges

Les questions qui ont motivées la définition du modèle 42 sont relatives à la notion de composants pour la modélisation et la simulation des systèmes embarqués hétérogènes, incluant les parties matérielles et logicielles. Les modèles pour le matériel sont intrinsèquement composants, du fait que le matériel est déjà partitionné en blocs. Pour le logiciel, plusieurs approches à composants ont été proposées. Cependant, un des challenges pour la modélisation des systèmes embarqués est de modéliser à la fois les composants matériels, les composants logiciels, ainsi que l'interaction entre eux, dans le même environnement.

Comme mentionné plus haut, une des utilisations possibles des modèles est de fournir un support pour la simulation. Plusieurs outils, qu'ils soient à usage académique ou industriel, ont été développés pour la simulation des systèmes embarqués. Ces outils ont montré leur efficacité à fournir des modèles de simulation de systèmes complexes comme les systèmes-sur-puce [Ghe06], réseaux-sur-puce [CCG+04], réseaux de capteurs [LFLL06], etc. De part notre expérience avec ces outils, nous somme arrivés à l'observation suivante : dans la plupart des outils, il existe une sorte de modèle à composants. Dans Ptolemy [EJL+03] par exemple, la notion de composants est claire et bien définie. Cependant, ce n'est pas le cas pour toutes les approches. Dans SystemC/TLM [Ghe06] par exemple, l'approche de modélisation est clairement modulaire, mais elle ne repose que sur des directives de conception de modèles. De plus, la plupart des outils ne proposent pas de moyens pour réfléchir sur les modèles préalablement ; leur intérêt est de fournir des modèles de simulation. Un axe de recherche serait de définir un environnement de modélisation rigoureux, indépendant de tout langage ou formalisme existant, qui peut être utilisé conjointement avec les outils existants.

Cet environnement de modélisation doit adopter une approche par composants à cause de la complexité des systèmes à modéliser. Pour cela, il doit fournir une définition claire de ce qu'est un composant pour les systèmes embarqués, et doit appliquer le principe du FAMAPSAP (*Forget As Much As Possible As Soon As Possible*). Ce que nous entendons par FAMAPSAP est l'analyse systématique des détails qui peuvent être encapsulés et les détails qui doivent être exposé par un composant.

Pour récapituler, les challenges suivants ont motivé la définition de 42:

- fournir un environnement, indépendant de tout langage ou formalisme, pour la modélisation par composants de systèmes matériels/logiciels.
- fournir un support pour une définition claire de la notion de composants, et aider à appliquer le FAMAPSAP.
- fournir un support pour l'intégration de modèles existants, issus d'outils hétérogènes, dans un environnement de prototypage virtuel ouvert.

1'.3 Résumé des Contributions

L'approche 42 est inspirée par Ptolemy : les système sont composés d'agents connectés entre eux pour échanger des données et un contrôleur pour déterminer ce qui se passe dans les connections, et comment les composants/agents sont activés. La grande différence avec Ptolemy est que les contrôleurs 42 sont décrits par des programmes utilisant un petit ensemble de primitives de base.

Nos contributions sont les suivantes :

- La définition (partiellement publiée dans [MB07]) d'une approche de modélisation basée sur les composants, ayant les propriétés suivantes : hiérarchie, indépendante de tout langage, spécification de composants en utilisant des contrats, exécutabilité des contrats, séparation du contrôle des données pour appliquer le principe du FAMAPASAP.
- Un ensemble assez riche d'exemples de modélisation pour montrer l'expressivité du modèle 42 pour la modélisation de différents *MoCCs* (voir chapitre 4).
- Un exemple de modélisation d'un système matériel/logiciel pour montrer l'utilisation des contrats 42 pour fournir des modèles de simulation de plateformes matérielles afin d'exécuter le logiciel embarqué (publié dans [BM09]).
- Un cas d'étude complet (publié dans [BMF09]) de l'utilisation de 42 avec des approches à composants existantes ; SystemC/TLM [Ghe06] dans notre exemple. L'exemple montre : l'intérêt de décrire des composants SystemC/TLM par des interfaces et contrats 42 ; la possibilité de générer de composants 42 depuis des langages existants ; la possibilité de simuler des systèmes décrits par des composants 42 et de composants SystemC/TLM.
- Un outil pour la conception est la simulation de modèles écrits en 42. Cet outil permet d'importer des composant existants issus d'autres approches, comme ça a été fait pour le cas de SystemC/TLM.

1'.4 Plan de la Thèse

Le contenu de cette thèse est inspiré des publication autour de 42 [MB07, BM09, BMF09]. Il est organisé comme suit :

Le Chapitre 2 "Background" est un ensemble de notions de base et de pratiques existantes dans divers domaines, qu'ils soient relatifs aux systèmes embarqués en particulier ou au génie logiciel en général. La plupart des idées présentes dans 42 sont inspirées des ces notions et pratiques. Ce chapitre sert de référence pour détailler l'ensemble de ces idées. Dans le reste de thèse, à chaque fois où nous rencontrons une de ces notions, nous mettons un pointeurs vers la section qui la détaille dans ce chapitre.

Le Chapitre 3 "Overview" est un aperçu des éléments de base de 42. Ce chapitre introduit la notion de composants et les moyens de spécifications que nous proposons pour les décrire. Ces spécifications peuvent être *implicites*, *explicites* (i.e., les *contrats de contrôle*), ou un mixe

entre les deux. Ce chapitre inclut aussi une brève présentation de quelques points adressés par 42 qui seront détaillés dans les chapitres suivants.

Le Chapitre 4 "Modeling Examples with 42 Components" est composé d'une série d'exemples de modélisation. Il décrit des *guidelines* pour l'écriture des composants pour quelques *MoCCs* et comment ces *MoCCs* peuvent être décrits par des contrôleurs 42. Le chapitre est partagé en deux grandes sections : la première section regroupe les exemples où les composants sont associés à des spécifications implicites. Dans la deuxième section, nous montrons comment utiliser les contrats de contrôle pour donner des informations explicites sur le comportement des composants 42.

Le Chapitre 5 "Formal Definition of 42" présente une définition formelle de 42. Nous y décrivons formellement les composants 42, l'assemblage de composants, les contrats, etc. Dans ce chapitre, nous décrivons aussi les programmes des contrôleurs au travers d'une sémantique opérationnelle, qui nous permettra par la suite de déduire le comportement de composants composites du comportement de leurs sous-composants et les programmes associés au contrôleur. Ce chapitre traite aussi les différentes notions de compatibilités entre composants, contrats, et contrôleurs.

Le Chapitre 6 "Exploiting 42 Control Contracts" est un chapitre dédié à l'utilisation des contrats de contrôle pour décrire le comportement des composants 42. En particulier, il donne des exemples de *MoCCs* où les contrôleurs peuvent être déduits directement des informations fournies par les contrats des composants et les dépendances de données entre les composants.

Le Chapitre 7 "Hardware Simulation and Software Execution" regroupe deux cas d'étude complets de l'usage de 42 dans le contexte du prototypage virtuel des systèmes-sur-puce. Dans un premier temps, nous montrons comment utiliser 42 pour écrire des prototypes virtuels de plateformes matérielles. Ces prototypes serviront par la suite de support pour le développement et l'exécution du logiciel embarqué. Le deuxième cas d'étude montre l'intérêt d'utiliser 42 conjointement avec des approches existantes. Nous avons choisi d'utiliser 42 avec SystemC/TLM vu que SystemC/TLM est l'un des standards dans l'industrie pour le prototypage virtuel des systèmes-sur-puce. L'intérêt de notre approche est de donner une description claire des composants TLM, ainsi que de fournir des modèles de simulation légers pour observer les synchronisations des composants TLM.

Le Chapitre 8 "Related Work" discute les choix que nous avons proposé pour 42 en comparaison aux approches qui existent déjà. Les approches décrites ne sont pas toutes dédiés aux systèmes embarqués. Pour chaque approche, nous donnons une brève présentation et nous discutons les similitudes et différences avec 42.

Le Chapitre 9 "A Tool for the 42 Component Model" décrit un outil que nous avons développé autour de 42. Cet outil implémente les éléments de base du modèle 42 et a permis de simuler l'intégralité des exemples qui sont présentés dans cette thèse.

Le Chapitre 10' "Conclusion & Prospects" conclut la thèse et présente quelques remarques et directions pour les travaux futurs autour de 42.

Suggestions pour le lecteur

- Le Chapitre 2, *Background*, décrit quelques notions et pratiques que le lecteur peut ne pas connaître. Vous pouvez ignorer ce chapitre dans le cas où vous connaissez déjà son contenu. Dan les autres chapitres, à chaque fois que nous avons besoin de détailler une des notions contenues dans le chapitre *Background*, un pointeur est mis vers la section concernées.
- Le Chapitre 3, *Overview*, doit être lu entièrement. Il donne un aperçu du modèle 42 et les éléments de base à retenir pour comprendre les exemples présentés dans la thèse.
- Si vous êtes intéressé, plus particulièrement par le *MoCC* synchrone, nous vous suggérons de lire la section 4.1.1 pour comprendre l'approche de modélisation des systèmes synchrones en 42. Ensuite, la section 4.2.1 pour comprendre comment les contrats sont utilisés pour décrire des composants synchrones. La section 4.2.2 présente la modélisation des systèmes synchrone multi-cycle. Enfin, la section 6.2.1 décrit comment générer des contrôleurs implémentant le *MoCC* synchrone depuis les contrats des composants.
- Si vous êtes intéressé, plus particulièrement par le *MoCCs* asynchrone, nous vous suggérons de lire la section 4.1.2 pour comprendre le principe de modélisation des *MoCCs* asynchrones en 42. Ensuite, la section 4.2.3 pour voir l'intérêt des contrats pour décrire les composants asynchrones. Enfin, la section 6.2.2 décrit la génération de contrôleurs implémentant des *MoCCs* asynchrones depuis les contrats des composants.

Chapter 1

Introduction

1.1 Embedded Systems and Their Development Cycle

Embedded systems are computer systems that are omnipresent in our everyday life. The domains where they may be used range from safety critical systems (avionics, nuclear plants, transportation, etc.) to consumer electronics (digital cameras, smartphones, etc.).

Compared to personal computer systems, the particularity of embedded systems lies in their hardware architecture. The hardware platform of embedded systems is often dedicated and differs from an application to another. The design choices of the hardware are guided by the functionality of the system and its required non-functional properties like energy consumption.

As hardware and software are tightly coupled, it is often unavoidable to design them together. However, finding an optimal solution is hard, because of the various parameters that should be taken into account, including computation speed, energy consumption, memory size, available surface, etc. Moreover, the hardware part may be available very late in the design cycle. Therefore, the design approach of embedded systems relies on *virtual prototyping*.

A virtual prototype is nothing more than an executable model of the embedded system. The use of models has numerous advantages: 1) When the specification language used for writing models is executable, one may predict the behavior of the future product by means of *simulations*; 2) When models are formally defined, there may be a possibility of asserting properties of the system by means of *formal validation*; 3) The final product may also be derived from the information provided by its model. This set of techniques is often referred to as *model-based development (design, etc.)*.

The difficulty of writing models comes from the intrinsic heterogeneity of embedded systems: they are composed of hardware and software, the hardware part may contain analog and digital circuits, the objects composing the systems are extremely concurrent and the concurrency model varies from pure synchrony to pure asynchrony, etc. In addition, heterogeneity appears during the design phase: the design of embedded systems involves some expertise from various domains including hardware engineering, control engineering, software engineering, etc. In each domain, there are several formalisms that answer crucial questions relevant to that domain. Each formalism has a proper understanding of the notions related to concurrency and timing aspects.

Hence, to cope with the heterogeneity of embedded systems, a modeling framework should encompass the modeling of the heterogeneous parts of a system. Moreover, it should be able to describe various types of concurrency models, communication mechanisms, timing aspects, synchronizations, etc. These notions are defined by the so-called *Model of Computation and Communication* (*MoCC*).

Another important notion is that of *components* for the design of embedded systems. Because of the complexity of the systems and the increasing time-to-market constraints, component-based approaches have seen their emergence. In the hardware industry, the notion of component is relatively old. *Intellectual Properties (IPs)* are off-the-shelf hardware components bought as physical blocks (to be plugged directly in the hardware platform), or as synthesizable specifications (to be integrated during the design phase). The notion of component in the hardware domain benefits a lot from the universal *MoCC* of synchronous circuits.

On the other hand, component-based approaches in the software industry are numerous. However, the notion of components for software is less generic than that for hardware because of the variety of *MoCCs* in which they may be used (threads, processes, event-driven, programs, etc.).

1.2 Component-based Virtual Prototyping and Challenges

The questions that motivated the design of the 42 approach are related to the notion of components in the modeling and simulation of heterogeneous embedded systems, including hardware and software parts. The models of hardware are intrinsically component-based, they benefit a lot from the hardware partitioning as blocks. For software, numerous component-based modeling approaches were proposed. However, one of the challenges for modeling embedded systems lies in modeling hardware and software as well as their interactions in the same framework.

As already mentioned, one of the uses of models is to perform simulations. For that purpose, plenty of academic and industrial tools were designed. They showed their effectiveness in simulating complex systems such as systems-on-a-chip [Ghe06], networks-on-a-chip [CCG+04], sensor networks [LFLL06], etc. Our experience with these tools raised the following observation: in most the tools, there exist (some sort of) a component model. In Ptolemy [EJL+03] for instance, the notion of components is clear and well-defined. However, this is not the case for other approaches like SystemC/TLM [Ghe06], where the modeling approach is clearly modular but relies on guidelines. Moreover, the only purpose of these tools is simulation. They do not propose a means for reasoning on models beforehand. For that purpose, some work has to be done in order to define a rigorous modeling framework independent from any language and usable jointly with the existing tools.

Such a modeling framework should be component-based because of the complexity of the systems to be modeled. For that purpose, it should have a clear definition of the notion of components for embedded systems and should enforce the FAMAPASAP (*Forget As Much As Possible As Soon As Possible*) principle. It should provide tools for the systematic analysis of the details that can be hidden vs the details that must be exposed by a component.

To summarize, the following challenges motivated the design of 42:

- provide a language-independent component-based framework for modeling hardware/software systems.
- provide support for a clean definition of components, and help enforcing the FAMAPASAP principle.
- provide support for integration of existing modeling and simulation tools in open virtual prototyping environments.

1.3 Summary of the Contributions

42 is inspired by Ptolemy: systems are made of agents connected to each other for exchanging data, and a controller determines what happens on the connections, and how the components/agents are activated. The essential difference with Ptolemy is that the controller is described as little programs in terms of more basic primitives.

The main contributions are the following:

- The complete definition (partially published in [MB07]) of a component-based modeling approach with the following properties: hierarchy, language-independent, explicit specifications with contracts, executability of contracts, separation of control and data to enforce the FAMAPASAP principle.
- A rich suite of examples, which demonstrate the expressiveness of 42 for modeling various *MoCCs* (see chapter 4).
- A simple hardware/software model to demonstrate the combined use of 42 contracts together with the embedded software for simulation (published in [BM09]).
- A complete case-study (published in [BMF09]) on the use of 42 together with an existing component-based approach, namely SystemC/TLM [Ghe06] for systems-on-a-chip. It demonstrates: the interest of the 42 interfaces and contracts; the possibility of generating 42 objects from an existing language; the possibility of simulating a system made of 42 components and SystemC/TLM components.
- A toolset for the design and simulation of 42 models. It is capable of importing existing components from other approaches. This has been applied to SystemC/TLM and Lustre.

1.4 Outline

The content of the thesis is inspired from the publications [MB07, BM09, BMF09] of the work dedicated to 42. The thesis is organized as follows:

Chapter 2 "Background" is a collection of basic notions. The reader may notice that most of the ideas presented in 42 are borrowed from the current practices of some approaches from distinct domains. We refer to these ideas during the presentation of 42.

Chapter 3 "Overview" is an overview of the basic elements of 42. It introduces the notion of components, and how they may be described by means of specifications. These specifications may be *implicit*, *explicit* (i.e., *control contracts*), or a mixture of both. It also includes brief presentations of some points tackled by 42.

Chapter 4 "Modeling Examples with 42 Components" is a suite of modeling examples. It illustrates some guidelines for modeling components and describing *MoCCs*. The examples are grouped into two sections. In the first section, we describe modeling examples based on implicit specifications of components. In the second one, we show how control contracts may be used in order to give explicit information about components.

Chapter 5 "Formal Definition of 42" presents the formal definitions of 42 elements. Namely, we define the notion of components, assemblies, the controllers implementing *MoCCs*, compositions, contracts, etc.

Chapter 6 "Exploiting 42 Control Contracts" illustrates the benefits of having control contracts associated with the components. In particular, it illustrates how some controllers may be deduced from the contracts of the components, and how the contracts may be executed.

Chapter 7 "Hardware Simulation and Software Execution" tackles two interesting uses of 42 in the context of systems-on-a-chip. First, we describe how 42 may be used in order to model virtual and executable prototypes of hardware platforms for the development of the embedded software. Second, we present first steps towards the formalization of SystemC/TLM with 42. SystemC/TLM being the de-facto standard in the industry for modeling systems-on-a-chip. However, it lacks in semantics.

Chapter 8 "Related Work" discusses the choices we made for 42 compared to various approaches, not all of them dedicated to embedded systems. We present each of them and discuss the differences/similarities with 42.

Chapter 9 "A Tool for the 42 Component Model" describes a prototype implementing the basic elements of 42. All the examples presented in this thesis were simulated with this prototype.

Chapter 10' "Conclusion & Prospects" is dedicated to concluding remarks and some directions for future work.

Suggestions for Readers

- Chapter 2, *Background*, is a collection of ideas that you may or may not know about. If you already know, you may skip it, we refer to the sections of this chapter when needed.
- Chapter 3, *Overview*, has to be read entirely. It provides an overview of all the aspects to be understood before reading the examples.
- If you are interested, in particular, in the synchronous *MoCC*, we suggest you read Section 4.1.1 for a global understanding of the modeling approach of synchronous systems with 42; then Section 4.2.1 to see how contracts may be useful for describing synchronous components; Section 4.2.2 presents the modeling of multi-cycle programs; finally, Section 6.2.1 describes how controllers implementing the synchronous *MoCC* are generated from the contracts.
- If you are interested, in particular, in the asynchronous *MoCC*, we suggest you read Section 4.1.2 for a global understanding of the approach; Section 4.2.3 to see how contracts may be useful for describing asynchronous components; finally, Section 6.2.2 describes how controllers implementing the asynchronous *MoCC* are generated from the contracts.

Chapter 2

Background

Introduction (En) *This chapter is presented as a collection of basic notions one should know about when reading the presentation of 42. We first introduce the most important discrete* MoCCs *for modeling concurrent systems. Then, we present virtual prototyping of the hardware in order to develop the software of systems-on-a-chip. 42 is a component based-approach, hence, we recall some aspects of components and their specification in software and hardware industry. At the end we give a brief presentation of the validation techniques used during the design of systems.*

Contents

Introduction (Fr) *La définition du modèle à composant 42 est inspirée de plusieurs notions existantes, que ce soit dans le domaine des systèmes embarqués, ou en génie logiciel en général. Dans ce chapitre, nous présentons les notions à connaître avant d'aborder la description de 42. Dans un premier temps, nous allons introduire des modèles de calcul discrets pour la modélisation des systèmes concurrents. Ensuite, nous présentons les techniques de prototypage virtuel de systèmes matériels, pour le développement de logiciel embarqué. Nous décrirons ensuite les notions de composants et de spécifications pour les composants logiciels et matériels. Nous terminerons par une description des techniques de validation de systèmes complexes.*

2.1 Models of Computation for Discrete Concurrent Systems

The design of embedded systems is subject to several constraints. The size of an embedded system is small compared to the size of a general purpose computer system. It often has limited resources in terms of computation speed, memory, energy, etc. The context in which an embedded system may be used varies from consumer electronics (phones, PDA, etc.), to safety critical systems (air plains, nuclear plants, etc.) in which bugs may cause considerable damage. All of these constraints make embedded systems complex, and impose careful decisions on their design.

2.1.0.1 The Design of Embedded Systems Requires Models

Due to their complexity, the design of embedded systems requires a modeling phase to understand the interaction between its various components, and the interaction of the system with its environment. Modeling may be used for:

- simulating the behavior of the future system.
- validating the system (testing, formal verification, etc.).
- analyzing the system with respect to timing, energy consumption, etc.
- designing a starting point of a *Model-Based Development* approach.

> *A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity which are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task* [Jan03].

A model is an abstraction of the entity being modeled, it exposes the relevant information we want to observe. Hence, an entity may be represented with various models depending on the use of the model. For instance, a model dedicated to performance analysis may be completely distinct from a model dedicated to functional validation.

2.1.0.2 Models of Computation and Communication

An embedded system is made of concurrent components that interact to expose the behavior of the whole system. The concurrency model varies from pure *Synchrony* (e.g., digital circuits) to pure *Asynchrony* (e.g., distributed systems).

To associate a complete system with its model, one has to define a model of each component of the system and define the concurrency model. That is, defining how components evolve together, and how they communicate. This amounts to providing the semantics of the *MoCC* (*Model of Computation and Communication*) governing the system.

The notion of time is relevant to the semantics of a *MoCC*. Time may be considered as continuous (where components are described with models based on differential equations), or discrete. In *Hybrid systems* the *MoCC* deals with the two notions. What follows will describe the main *discrete MoCCs*.

2.1.1 Synchronous Models and Languages

Synchronous languages [BCE+03] are well suited to design *reactive systems* [Hal92, HLR92]; that is, systems with permanent interaction with their environment. A reactive system repeatedly

gets its inputs (reading sensors for instance) from the environment and produces the outputs (controlling an actuator for instance). A reactive system should compute fast enough in order not to miss relevant events (inputs).

The Notion of Clocks In synchronous systems, there is a notion of *logical clock* that indicates when the system should read the inputs to compute the outputs. At each clock tick, the computation of the outputs involve the parallel reaction of all the components of the system. Each of them computes its outputs based on the inputs it is provided with. In a real system, the clock would be associated with the arrival of some event (from the environment) that may, or may not, be related to physical time. For instance, the timer signal in a computer system (time), a train reaching a beacon (distance), etc., may define a clock for a synchronous system.

Communication The communication mechanism between synchronous components is the *synchronous broadcast*. At each clock tick, a component sends some values through it outputs and does not need to know whether one or several components are waiting for these values. Sending is non-blocking.

Instantaneous Computation Synchronous components are supposed to read inputs and compute outputs instantaneously (i.e., during the same clock cycle). In practice, instantaneous computation of the outputs is not feasible. However, the requirement for reactive systems is that the reaction is sufficiently fast in order not to miss incoming events. The code produced by synchronous languages compilers is quite simple, it may be used for *Worst Case Execution Time* analysis [Rin00]. Hence, it is possible to check that the system reacts to all incoming events if we have some knowledge about their arrival frequency.

Interesting Uses Besides the design of reactive systems, several techniques were developed around the synchronous languages for the purpose of modeling, simulation and validation of safety critical systems.

The modeling of embedded systems often requires asynchrony and non-determinism. It is well known that synchronous formalisms can be used to model asynchronous parallelism [Mil83]. In fact, the synchronous paradigm may be used to model all kinds of intermediate behaviors, between pure synchrony and pure asynchrony.

Synchronous languages propose some operators to prevent a component from reacting (e.g., *Clocks* in Lustre and Signal, the *suspend* statement in Esterel, *activation conditions* in SCADE, etc.). Thanks to such operators, one can decide that components do not execute at the same clock cycles, hence, they are asynchronous (see 4.2.2.8 for illustration). The most interesting instance of this principle is the so-called *quasi-synchronous* approach [CMP01], to describe systems made of several processors that are not explicitly synchronized. Their respective clocks may differ, but not in a completely unknown way. These systems are modeled by a quite liberal constraint on the clocks, namely: *there are never more than two ticks of one clock between two ticks of the other one.*

The purpose of modeling asynchrony by means of a synchronous formalism is to benefit from the validation approaches developed around synchronous languages [HM06]. Such an approach has been applied to several case studies from the industry [JHR+07, GG03]. Moreover, recent work have been devoted to the use of synchronous formalisms for the analysis of non-functional properties such as energy consumption [LFLL06], and performance analysis in stream processing systems [AM10, ALM10].

In the sequel we present some languages for the design of synchronous systems.

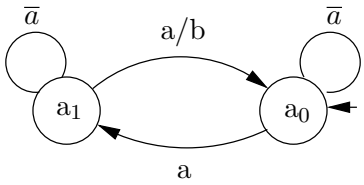


Fig. 2.1: A Boolean Mealy machines

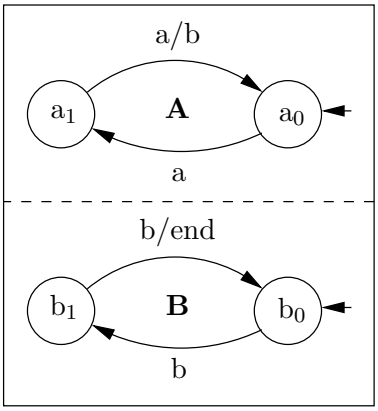


Fig. 2.2: The modeling of a modulo-4 a-counter with Boolean Mealy machines

2.1.1.1 Automata-Based Models

We present in this section an example of modeling synchronous systems by means of *Mealy machines*. Mealy machines are automata (precisely *transducers*) in which the computation of the outputs depends on the state of the automaton, and the current values of the inputs. In the rest of the section, we borrow the syntax from the synchronous language *Argos* to describe the example. In the category of automata-based languages, one may refer to SyncCharts [And04] also.

Figure 2.1 illustrates a *Boolean Mealy machine*: a Mealy machine where the inputs (`a` in the figure) and the outputs (`b` in the figure) are of Boolean type. Each transition of the automaton is labeled with `inputs/outputs`. The behavior of the machine is as follows:

- At state a_0, if it receives `a` then it does not emit `b`, and moves to state a_1. Otherwise, if it receives $\overline{a}$ (which stand for he negation of `a`), it does not emit `b` and does not change state.
- At state a_1, if it receives `a` then it emits `b`, and moves to state a_0. Otherwise, if it receives $\overline{a}$, it does not emit `b` and does not change state.

The loops over the states that does not emit some outputs are often omitted (see Figure 2.2).

A Modeling Example Figure 2.2 illustrates the modeling of a modulo-4 a-counter[1] with two Boolean Mealy machines. `A` was described previously in Figure 2.1. Notice that the loops over the states are made implicit. `B` exposes the same behavior, but it has `b` as input and `end` as output. As `b` is the output of `A`, there exists a communication between `A` and `B`.

At each step, each component (i.e., `A` and `B`) takes one transition depending on its inputs. The outputs of the transition are broadcasted to all the components running in parallel. The behavior of the parallel reaction of the two automata is as follows:

Initially, the global state is a_0b_0 which encodes 0. The first occurrence of `a` moves the global

[1]The example is inspired by the presentation of Argos in [MR01]

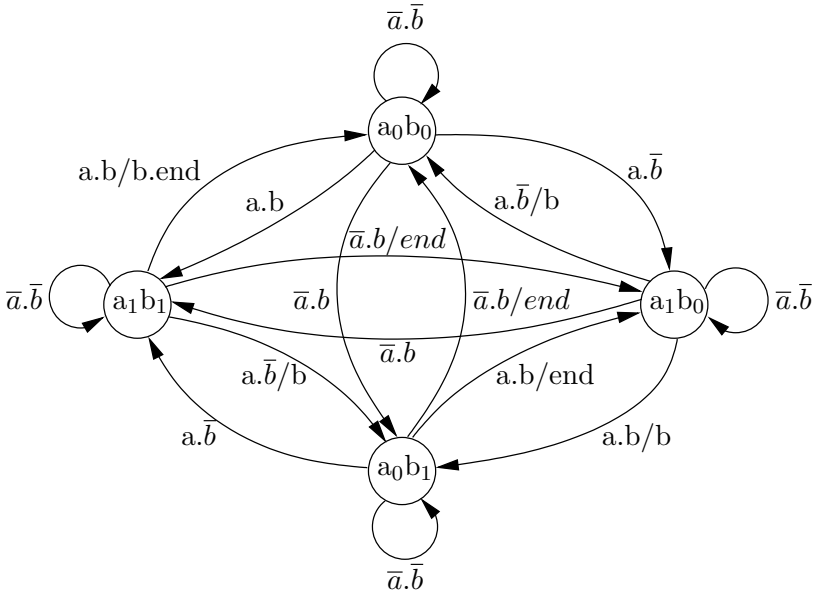


Fig. 2.3: The synchronous product of the Mealy machines A and B of Figure 2.2

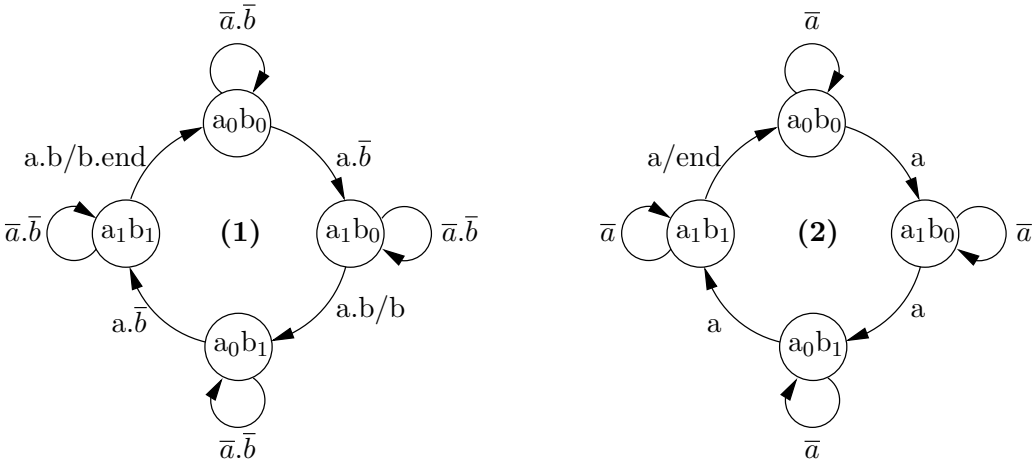


Fig. 2.4: **(1)**: Encapsulating and **(2)**: hiding the signal `b` in the product of Figure 2.3

state to $\mathtt{a}_1\mathtt{b}_0$ which encodes 1: `A` takes a transition and changes state; `B` takes the implicit transition and does not change its state. The second occurrence of `a` moves the global state to $\mathtt{a}_0\mathtt{b}_1$ encoding 2: `A` changes its state and emits `b`. At the same step, `B` takes its transition and changes its state because its input `b` is emitted.

The Synchronous Product Figure 2.3 describes the synchronous product of the Boolean Mealy machines of Figure 2.2. It is also a Boolean Mealy machine, where the inputs (resp., outputs) are the union of the inputs (resp., outputs) of the two components. Each transition of the product corresponds to exactly one transition of each of the components. Notice that the synchronous product does not make any synchronization between components.

Encapsulation The encapsulation is parameterized by a set of signal names. It is used to restrict the scope of a signal, and to force synchronization between components. For instance, the signal `b` of Figure 2.2 which is the output of `A` and the input of `B` may be used to synchronize `A` and `B`. In this case, `b` becomes a local signal.

Figure 2.4 illustrates the steps of encapsulating the synchronous product of Figure 2.3. Firstly, we remove some transitions (illustrated by **(1)** in Figure 2.4). Secondly, as the encapsulated signals are local, we hide their names in the labels (illustrated by **(2)** in Figure 2.4).

The transition that are removed during the encapsulation are those that do not fit in the two following rules:

- A local signal which is supposed to be present has to be emitted in the same reaction. Transitions like `a.b`, $\overline{a}$`.b`, etc., are removed.
- A local signal that is supposed to be absent should not be emitted in the same reaction. Transitions like `a.`$\overline{b}$`/b` are removed.

2.1.1.2 Synchronous Programming Languages

Among the languages designed for synchronous programming, one would refer to the imperative style language Esterel [BG92], and the declarative languages Lustre [HCRP91], and Signal [GG87].

In this section, we present the synchronous language Lustre, to which a lot of work has been devoted, in order to develop techniques for program verification [RHR91, GH06, Gon07], sequential and distributed code generation [Gir94], etc.

```
node integrator(i : int)
returns (o : int) ;
let
 o = i -> pre(o) + i ;
tel.
```

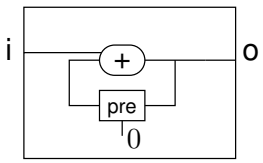


Fig. 2.5: An integrator written in textual Lustre and its graphical representation

Figure 2.5 is an example of a program written in Lustre, and its graphical representation as it would be described in the commercial tool SCADE[2].

A program written in Lustre is called a `node`. The interface of a node exposes the set of its inputs and outputs (`i` and `o` in the example). Lustre is a data-flow declarative language; this means that the internal variables, the inputs and the outputs are flows of values.

When there is only the global clock (mono-clock synchronous programs), at each instant of the global clock, each flow is given a value. The value of a flow is defined as a function of other flows. The function may be a basic operator (i.e., +, -, *, etc) or a node; which means that a node may be used to design a new node in Lustre.

Cyclic Dependencies and the Delay Operator Because of the dependencies of the flows, at each instant, there is an order in which the values of the flows are computed. The architecture of a Lustre program as it would be described by its graphical representation, defines a partial order on the dependencies of the flows. Programs containing cyclic dependencies are rejected by the Lustre compiler as there is no way to compute an order of computation. Figure 2.6 illustrates the case where there exists a cycle between the flow `o` and itself. Such a program is rejected.

The `pre` operator introduces a delay. In Figure 2.5, the statement `o = i -> i + pre(o)` means that `o` takes the value of `i` at the first instant (`->` is the initialization operator), and for all the instants $\mathtt{n}_k$ such that `k > 0` the value of `o` is the value of the input `i` plus the `previous` value of `o` (i.e., the value of `o` at instant $\mathtt{n}_{k-1}$). This statement exposes a cycle (`o` depends on itself),

[2] SCADE is a commercial tool for the design of safety critical systems; it is based on Lustre

```
node cycle(i : int) returns (o : int) ;
let
 o = i -> o + i ;
tel.
```

Fig. 2.6: A cyclic dependency in a Lustre program: the flow `o` depends on itself

but is still correct because of the delay. The value of `o` depends on its previous value. In Lustre, every cycle must be cut with a `pre` operator.

Clocks and Clock Manipulation Operators Lustre may be used to describe systems with several clocks in addition to the global clock. Clocks are associated with flows (i.e., the variables of a Lustre program) to define the presence or the absence of a value of the flow at a given instant. In Lustre any Boolean flow may define a clock for another flow.

X	0	1	2	3	4	5	6	7
CLK	true	false	true	false	false	false	true	true
Y = X when CLK	0		2				6	7
Z = current Y	0	0	2	2	2	2	6	7

Fig. 2.7: The `when/current` operators in Lustre

Lustre provides some primitives to manipulate clocks. The operator `when` is a *sampling operator*. It is used to define a slower flow from a faster one. In Figure 2.7, the declaration `Y = X when CLK` says that the flow `Y` takes the value of the flow `X` each time the flow `CLK` has the value true. Hence, `CLK` is the clock associated with the flow `Y`.

The operator `current` is a *projection operator*. The operator is used to get a faster flow from a slower one. For instance `Z = current Y` says that the flow `Z` takes the value of the flow `Y` if `Y` is present, otherwise it takes the value of `Y` the last time it was present.

2.1.2 Asynchronous Models

When modeling embedded systems, there is often a need for describing systems where components evolve asynchronously. Contrary to synchronous ones, asynchronous systems are systems where there is no common clock that may be shared between components. Asynchronous systems range from threads on a mono-processor systems to large scale multi-computer systems.

On a mono-processor system, several processes or threads may run (in parallel) thanks to time-sharing schedulers, and may access a shared memory. On multi-computer systems, processes run in real parallelism, and may synchronize by message-passing.

2.1.2.1 Modeling Asynchronous Behaviors without Communication

The general modeling of asynchronous systems relies on *interleaving* semantics. That is, for a whole system, an execution step corresponds to one execution step of one of its components.

Figure 2.8 describes two automata **(A)** and **(B)** that may model the behavior of processes for instance. The states of the automata model the state of the processes. Each transition corresponds to an action (or a set of actions) of the process and is considered to be atomic. In other words, nobody running in parallel can observe the internal states of **(A)** during action `a`.

On the figure, **(A×B)** describes the asynchronous product of **(A)** and **(B)**. The asynchronous product produces the states that are potentially reachable when the two automata are run in

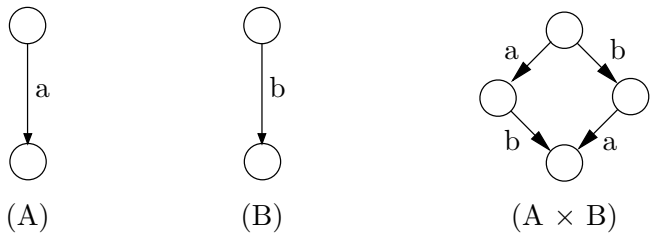


Fig. 2.8: Asynchronous models without communication

parallel.

A Note on Granularity Figure 2.9 recalls the importance of atomicity in such asynchronous models, based on interleaving semantics. Suppose we replace the automaton **(B)** of Figure 2.8 with the automaton **(B')**, where the transition `b` is no longer atomic, it is split into to atomic actions `b'` and `b''`.

Now, the global behaviors are those of automaton **(A×B')**, in which new states appear. The `a` transition from state `X` means that in **(B')**, the intermediate state between `b'` and `b''` is observable by **(A)**.

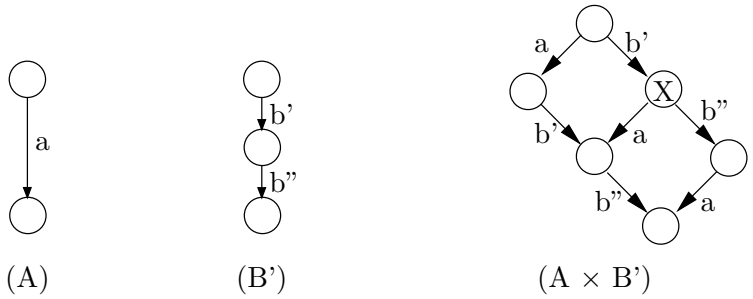


Fig. 2.9: Fine granularity of atomic executions exposes more states

Choosing the granularity of atomic transitions is an intrinsic modeling problem. When modeling the behaviors of two threads on a mono-processor with a preemptive scheduler, the only possible choice is to consider the atomicity as given by the execution platform: instructions in the processor are atomic (non interruptible by the scheduler). Hence a thread is described by a detailed automaton, with explicit states between machine instructions. The model is appropriate for checking parallel programs that use low-level synchronization mechanisms like semaphores of atomic read-write machine instructions.

However, in high-level models, the notion of atomicity may be of a coarser granularity. In transaction-level models (see Section 2.2.1.1), typically, the granularity reflects the fact that we do not need to observe the precise interleavings of the component behaviors. We only need to observe the interleavings at a granularity given by the explicit synchronizations between the components.

2.1.2.2 Modeling Asynchronous Behaviors with Shared Memory

Figure 2.10 illustrates the modeling of communicating asynchronous processes. As the processes may not be alive at the same time, they communicate through a shared Boolean variable `x` initialized to `false`.

The process `A` assigns `true` to the variable `x`. The process `B` may not execute its transition until `x` has the value true.

The states of the asynchronous product include the state of the memory location relevant for the two processes. Here, the variable `x`.

Due to the communication between the processes, the left path of the product (dashed lines) is not possible. Notice that we still rely on interleaving semantics, but some of the states in the asynchronous product are no longer reachable.

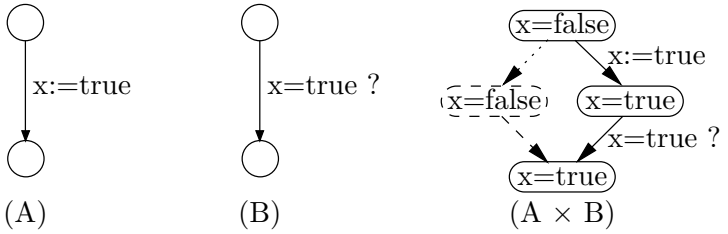


Fig. 2.10: Asynchronous models with shared memory

2.2 Virtual Prototypes of Hardware Platforms for Software Development

The development of embedded systems has to take into account the design of the hardware part, and the software that may run on the hardware platform.

Modeling the hardware platform is an important phase in the design of an embedded system. There are plenty of tools that allow for the design of virtual prototypes of the hardware at various levels of abstraction.

Beside the fact that virtual prototypes of the hardware may be used for accurate simulation, design verification, and automatic generation of the hardware platform, software developers need virtual prototypes of the hardware to start developing the embedded software. The fact is that the hardware part of embedded systems is often changing from a system to another. Hence, software developers may not rely on programming models to write the software, they need executable prototypes of the real hardware on which the software would run.

In this section we discuss some tools for the modeling of the hardware and how such models would be used as a virtual execution platform for software development.

2.2.1 Modeling Hardware Platforms

The behavior of the hardware may be modeled at several abstraction levels. At each abstraction level, timing aspects are defined precisely, approximately, or even inexistent. An abstraction level also defines the notion of granularity in the observable simulation states and the data exchanges.

Low Level of Abstraction Models HDLs (*Hardware Description Languages*), allow for the modeling of the hardware at RT-Level (*Register Transfer Level*). Tools like Verilog [ver] and VHDL [vhd92] propose a textual language to specify hardware. Programs written in such languages may be simulated with a precise notion of time; they are also used as input to automatic generation of the hardware layer. The synchronous languages Lustre [RH92] and Esterel [Ber92] together with SystemC [sysa] may also be used to describe hardware at the RTL level.

High Level of Abstraction Models Other approaches target the modeling of hardware at higher levels of abstraction. They may abstract away from the details that do not matter with regards to the modeling purposes. For instance, AADL (*Architecture Analysis and Design Language*) [FLVC05] is a component-based approach for modeling (not only) hardware architectures. The details exposed by the models written in AADL allow to perform architecture exploration and some timing analysis related to data flows.

SysML [Wei08], an extension of UML, has also been proposed to tackle specific modeling problems. It may be used for modeling of hardware at a high level of abstraction. For the moment it is not executable, and the semantics is not formalized, so it is of limited use.

Metropolis [BWH+03] is a formally defined methodology for the development of embedded systems. Abstract hardware models may be designed with Metropolis; they provide an API for the software development.

In the same category one can refer to approaches like SystemC/TLM [Ghe06], SystemVerilog [SYSb], SpecC [GZD+00], etc.

What Kind of Models do Software Developers Need Instead of waiting for the hardware platform to be manufactured, software developers use a virtual and executable prototype of the hardware to start developing the software. The main motivation is to decrease time-to-market.

It is possible to use the RTL model as an executable model of the hardware. However, in addition to the late availability of the RTL model, the hardware/software co-simulation is too slow to allow for effective development of the software. The low co-simulation speed is due to the quantity of details exposed by the RTL model.

To cope with the slow simulation speed, new abstraction levels have emerged. For instance, *Transaction Level Modeling (TLM)* [Ghe06] uses a component-based approach, in which hardware blocks are described by *modules*; the communication between the hardware blocks over the real bus is abstracted by the so-called *transactions*.

TLM allows several abstraction levels, from cycle-accurate to pure functional models. They are better suited than RTL models for early development of the embedded software, because the high level of abstraction allows a faster simulation.

In the sequel we present the *Programmer's View* (PV) abstraction level of TLM by means of an example written in SystemC/TLM.

2.2.1.1 TLM Programmer's View with SystemC

SystemC [sys06] is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to purely functional models. It comes with a simulation environment, and has become a *de-facto* standard. SystemC offers a set of primitives for the description of parallel activities representing the physical parallelism of the hardware blocks. The TLM/PV level of abstraction is implemented in SystemC by defining specific TLM libraries and templates. TLM 2.0 has been standardized by the Open SystemC Initiative (OSCI).

The success of SystemC comes from its C/C++ part: it is widely known, it is a general-purpose programming language, so there are no restrictions on what the designer can write, and it makes it possible to build tools for the co-simulation of SystemC code with C, assembly code, RTL models, etc. The dark side of the picture is that SystemC models are quite hard to analyze formally (SystemC, being based on C++, has no formal semantics, and SystemC models may implement relatively simple things by complex integer manipulations, which make simple properties undecidable in the general case). Therefore the main activity with a SystemC model is simulation.

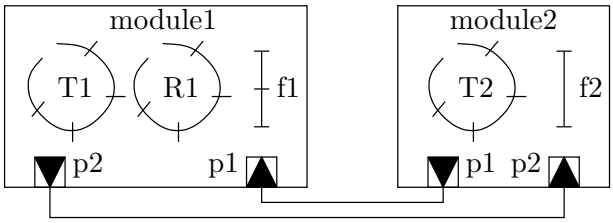


Fig. 2.11: Example of SystemC/TLM (architecture)

SystemC Modules A TL-Model is based on an *architecture*, i.e., a set of components that expose *ports*, and connections between them, as shown on Figure 2.11. The components, also called *modules*, represent some physical entities that behave *in parallel* in the real system to be modeled (typically: a bus, a CPU, a memory, etc.).

Components communicate through the ports by means of *transactions*. The direction of the arrow shown on a port determines the role of the port during transactions. On Figure 2.11, `p2` in `module1`, and `p1` in `module2`, are *initiator* ports of the transactions; `p1` in `module1`, and `p2` in `module2`, are *target* ports.

A Module is Composed of Threads and Functions The behavior of a component is given by a set of *threads* (represented by circles with *steps* on Figure 2.11), and a set of *functions* (represented by straight lines with steps), both programmed in full C++ (see Figure 2.13). `module1` has two threads `T1` and `R1`, and one function `f1`, while `module2` has a single thread `T2` and a function `f2`. The threads are *active* code, to be scheduled by the global scheduler; the functions are *passive* code, offered to the other components, and that will be called from a thread or function of another component; the functions are attached to the target ports of the module (`f1` is attached to `p1` in `module1`). Inside such a module, the processes and the functions may share *events* in order to synchronize with each other. Events can be notified, or waited for. In `module1` (Figure 2.13) there are three events `e1, e2, e3`.

The SystemC Scheduler All the threads are managed globally by a non-deterministic scheduler. The SystemC scheduler is *non-preemptive*: a running thread has to yield, by performing a wait on an event or on *time*. For instance, for the thread `T1` of `module1`, the only point where the thread yields is `wait(e1)`. The execution of `a++; e2.notify(); a++; e3.notify();` is therefore atomic.

Figure 2.12 [3] is a sketch of the algorithm of the SystemC scheduler. After the elaboration phase, where the set of modules and communication channels are instantiated, the scheduler executes the phases presented below; a more concise description would be found in [sys06, Cor08]:

EV Is the evaluation phase. The scheduler executes the eligible processes in a non-deterministic order. Processes yield back control by waiting for an event or time to elapse. Notified events may render some processes eligible. These processes are executed during the same phase. When there is no eligible process, the scheduler enters the update phase (UP).

UP During the evaluation phase, SystemC objects may request to be activated at the update phase. For instance, some communication channels may request the scheduler to call their *update* function to update their values. Some events may be notified during this phase. Then the scheduler re-executes the EV phase.

[3] The figure is taken from [Bou07]

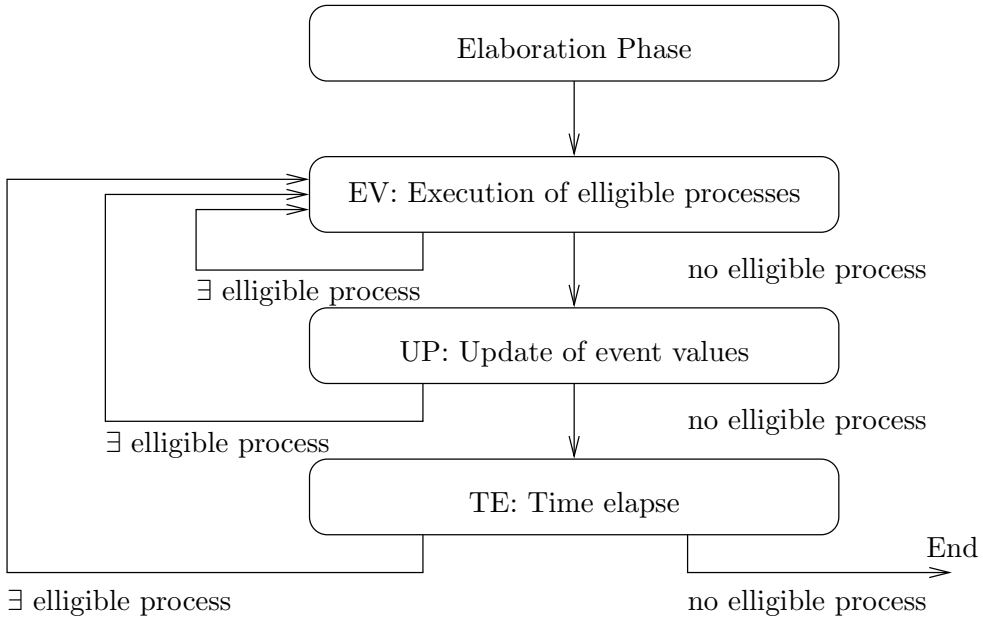


Fig. 2.12: The SystemC scheduler

TE When no process still eligible after the update phase, the scheduler lets time elapse until reaching the earliest delay waited for by processes. The corresponding processes are made eligible, and the scheduler enters a new evaluation phase.

Some TLM Guidelines According to the TLM-PV guidelines, communications between modules cannot use the event mechanism, because this would be meaningless w.r.t. the physical parallelism to be modeled. The only possible communications are called *transactions*, implemented by blocking function calls; the link between a caller and the callee is established through the architecture. On Figure 2.13, the thread `T2` of `module2` initiates a transaction on its port `p1` (written `p1.f1(c)`). This is a call to the function `f1` in `module1` (which is attached to the target port `p1` of `module1`), because the initiator port `p1` of `module2` is connected to the target port `p1` of `module1`. When the call is executed (`T2` being running), the control flow is transferred to `module1`, until `f1` terminates; then the control flow returns to `module2`, and the execution continues until the next yielding point (`wait(e4)` in the example). Since function `f1` in `module1` waits for the event `e3`, it yields if `e3` is not present; this means that thread `T2` in `module2` may yield because of a `wait` statement in the function it has called in another module.

An example atomic sequence is the following: the scheduler elects thread `R1`, which is at line 12; it executes `b++` and then calls `f2` via the port `p2`; the body of `f2` in `module2` is executed entirely, and the control returns to `module1`; the thread `R1` loops, executes `b++` at line 11, and stops on `wait(e2)` at line 11.

2.2.2 Executing Embedded Software on a Virtual Prototype of the Hardware

As explained previously, hardware virtual prototypes are used by software developers to write the embedded software. The virtual prototype provides an execution platform for the software. This means that the software should execute together with the model of the hardware.

Suppose we (as software developers) are provided with the virtual prototype of the hardware platform described by Figure 2.14. The virtual prototype contains a model of the memory, the

```
1  void module1::T1(){
2    int a = 0;
3    while(true){
4      wait(e1);
5      a++; e2.notify();
6      a++; e3.notify();
7   }}
8  void module1::R1(){
9    int b = 0;
10   while(true){
11     b++; wait(e2);
12     b++; p2.f2(b);
13  }}
14 void module1::f1(int x){
15   cout << x ;
16   e1.notify();
17   wait(e3);
18 }// ** module1 **
```

```
19 void module2::T2(){
20   int c;
21   while(true){
22    c++;  p1.f1(c);
23    c++;  wait(e4);
24   }
25  }
26
27 void module2::f2(int x){
28    cout<< x ;
29    e4.notify();
30 }// ** module2 **
```

Fig. 2.13: Example of SystemC/TLM (code)

LCD (*Liquid Crystal Display*), the BUS, and the CPU. Once the software is written, we need to execute it for simulation and debugging.

To make the software execute with the hardware model, engineers in the industry rely on two approaches. In what follows we describe the two approaches.

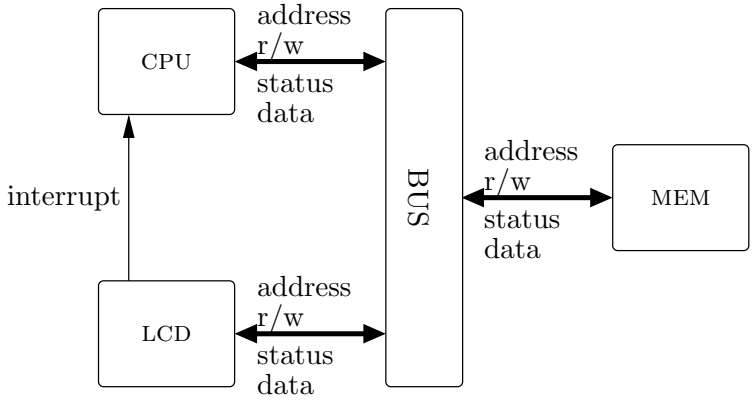


Fig. 2.14: Example of hardware platform

2.2.2.1 Native Wrappers

Using a native wrapper, the embedded software is wrapped into a piece of code. The goal of the wrapper is to intercept the decisions of the software. The software decisions we want to intercept are those related to the communication of the software with the hardware components. For instance, we need to know when the software writes into the memory, what data to be written, and to which address.

The wrapper is provided by the designer of the virtual prototype; it is part of the model of the processor. The wrapper offers to the software developer primitives related to the communication with the hardware. For instance, it may provide primitives like `write_mem(int address, int data)` to write into the memory, `read_mem(int address, int* data)` to read from it, etc.

The wrapper primitives are implemented so that to reflect the decisions of the software on the virtual prototype of the hardware components. For instance, when the software calls `write_mem(a,d)`, the wrapper engages in a set of actions in order to write the data `d` at the address `a` of the component modeling the memory.

To simulate the whole system, the wrapped software code together with the hardware virtual prototype are compiled into a binary code. The binary code is directly executed by the processor of the host machine on which the simulation will run.

Once we are happy with the software implementation, we integrate it into the real hardware platform. To do so, the wrapper primitives are re-implemented as accesses to the registers of the target processor with additional information on memory-mapping; the software together with the wrapper are cross-compiled into the target processor binary code; then, the binary code may be integrated in the hardware platform.

2.2.2.2 ISS-Based Software Execution

An *Instruction Set Simulator* emulates the behavior of a specific processor when executing a binary code. The ISS is able to interpret the complete *instruction set* of the processor, and maintains a set of variables that corresponds to the registers of the processor.

Contrary to native wrapper simulation, ISS-based simulation requires the software to be cross-compiled into the binary code of processor of the hardware platform. The resulting binary code, is given as input to the model of the CPU in the virtual prototype. The CPU model is in fact the ISS.

During the simulation, the model of the hardware (including the ISS) is executed by the host machine; the ISS interprets the binary code of the software and reflects its behavior on the hardware model.

When the simulation gives good results, the binary code executed by the ISS, is integrated directly into the real hardware platform. The embedded system would expose the same behavior as the simulation model if the virtual prototype is faithful to the hardware platform.

2.3 Components and Contracts

The notion of a *component* for embedded systems has been discussed for some years now, and there are a lot of proposals. The main motivations are the following: as time-to-market decreases, it becomes unavoidable to *reuse* a lot of previous work when designing new systems.

Reusing parts of a previous system requires that these parts be properly defined as *components*, equipped with some form of a *specification* (informal or formal). The *specification* groups all information needed for using the component, without knowing in details how it is built.

2.3.1 Components

A typical embedded system is built from hardware and software. Component-based design of embedded systems should take into account the hardware part and the software part. Even if there is clear definition of what a software (resp., hardware) component is, mixing software and hardware components may not be an easy task.

Components for Hardware The notion of components in hardware is very old. There is a very clear definition of a component in the hardware industry, at the RTL level. For instance,

IPs (for *intellectual properties*) are off-the-shelf hardware components. An IP may be used as a physical hardware block or integrated during the modeling phase as a specification written in any HDL (*Hardware Description Language*). The synchronous semantics of the RTL description for circuits is what makes the definition of components widely portable.

Components for Software The notion of components for software is less generic than the notion of hardware components, because the semantics of software compositions are more diverse than the semantics of hardware compositions (at least at the RTL level, where there is a single semantics).

Several approaches for component-based software development have been proposed, not all dedicated to embedded systems. From one approach to another, there are a lot of distinctions; the interface of a component differs, the execution model may be distinct, etc.

An interesting property of component-based models is the notion of *hierarchy*. A model is hierarchic if a composed component (made of an assembly of other components) is not distinguishable from a basic one. Fractal [BCL+06a] and SOFA [PBJ98] component models are hierarchic. Other approaches are not hierarchic. For instance, CORBA, a software architecture for the deployment of heterogeneous components (distinct programming language and execution platforms), EJB (*Enterprise Java Beens*) [EJB03] and CCM (*CORBA Component Model*) [CCM]. These approaches are not hierarchic in the sense that they do not offer means to encapsulate an assembly of components in order to build one component exposing the same interface as a basic component.

2.3.2 Specifying Components

Component-based approaches facilitate the design of systems by reusing parts of previous designs. Moreover, the idea behind using components, is to be able to connect components to form the system. Even if individual components are validated by intensive use and testing, putting them together may raise some problems. Specifying components may help detecting such problems.

The *specification* of a component may group all the information we need, including both *functional* and *non-functional* aspects, like timing performances, or energy consumption. The CBSE *Component-Based Software Engineering* community identifies four classes of specifications (or contracts) [BJPW99]:

Basic or syntactic contracts describe the set of possible operations the component may perform. IDLs (*Interface Description Languages*) define basic contracts. IDLs are adopted by almost all component-based approaches to allow communication between components. The following code is an example of specifying a Java application with *Java IDL*. The `module` keyword corresponds to a package in Java. The description of the interface `fifo` declares its syntactic contract. The contract declares the operations provided by `fifo` together with their signature.

```
module buffers
{
  interface fifo
  {
    string get();
    void put (string);
  };
};
```

Behavioral contracts enable the description of the behavior of component when it is asked to perform an operation. Behavioral contracts deal with the *pre* and *post* conditions, and invariants. First introduced in the Eiffel programming language [Mey97], behavioral contracts have gained much interest and were introduced in other approaches like iContract [Kra98] (a design-by-contract tool for Java developers), and OCL [WK03] (*Object Constraint Language*), which is part of UML (*Unified Modeling Language*).

Synchronization contracts deal with some dependencies between service calls of a component. They enable the description of the correct sequences of service calls, concurrency, mutual exclusion, etc. The notion of protocols in the object programming language PROCOL [VDBL89] is an example of synchronization contracts.

Quality of service contracts are used to describe non functional behaviors. They provide information related to the quantification of some parameters of the service like the response time, energy consumption, quality of an image, etc. This information may be used by the interacting components for negotiation purposes as it would be done in Qinna [TBO05]. Qinna is a component-based QoS architecture designed to manage QoS issues.

Specification Languages for Software Components Component-based approaches insist on the fact that the implementation details of a component should be invisible. Components are considered as *black boxes*. Hence, an abstract description of components is required.

A lot of work has been advocated to the design of specification languages for various component-based approaches. *Behavior protocols* [PV02] for instance, were designed to specify the behavior of a component by defining the sequence of method calls emitted and accepted by a component. Initially, behavior protocols were designed for the component model SOFA, but they were also ported to the component model FRACTAL.

Interface automata [dAH01] were designed in the same sense. They equip components with an automata-based description that defines how components react to their environment. The transitions in the automata are labeled with inputs, outputs, and internal actions.

Approaches like *session types* [VVR06] express multiple protocols for one component, each protocol being related to a specific component interface. The protocols are considered as types.

The specifications described as behaviors are usually exploited to determine some properties over a set of components. Based on the specifications, one can check whether composing components exposes a correct behavior or not (i.e., *compatibility*) or whether a component may be replaced by another one in an assembly without changing the global behavior (i.e., *substitutability*).

2.3.3 Design By Contract

Design-by-Contract is a design approach first introduced by Bertrand Meyer in the Eiffel programming language [Mey97, Mey92]. It insists on the fact that the design of software modules should imperatively include the specification phase. The specification consists in documenting the methods of the module with contracts before writing them.

If we rely on the classification given above, the contracts should describe behavioral aspects:

- **pre-conditions** of a method call: A property that should hold when the corresponding method is called.
- **post-conditions** of a method call: A property that the component guarantees, after being called and having executed successfully.
- **class invariant:** A global property on the state of the module, that should hold before and after a call to one of its methods.

In addition, the contract declares the *exceptions* that may be thrown when a contract is violated.

Design-by-Contract Vs Defensive Programming Design-by-contract aims at making a clean separation between the specification of a module and its functional behavior. Once the contracts are established, the designer of a module should concentrate on the implementation part, not on checking the consistency of the contracts.

Writing additional code to check whether the contract of a method is violated is a practice often referred to as *defensive programming*. This practice negatively affects the readability of the code and its performance during execution.

In design-by-contract, checking the consistency of the contracts is performed by the underlying execution platform. Once the systems are validated, notice that the contracts may be removed to gain in efficiency without affecting the functional behavior of the system.

2.4 Validation

Validation is the process of ensuring that a system meets the requirements of the designer. The designer requires that the implementation of a system conforms to some properties. Usually, these properties are established before the implementation phase. For instance, a component may be validated individually by checking whether its implementation meets its specification.

Validation techniques are grouped into two categories: approaches that rely on the execution of a system to validate (simulation and runtime verification), and approaches that do not require the execution of the system (static verification).

2.4.1 Formal Specification of Properties

The properties of a system one would verify are mainly grouped into two classes:

Safety properties stipulate that *something wrong will never happen*. For example, a property expressing that a variable `X` will never take the value `0` during the execution of a program.

Liveness properties stipulate that *Something good will eventually happen*. For example, a property expressing that when a message is sent, it will be received.

Such properties may be expressed formally in terms of *temporal logics*. Temporal logics are formalisms in which complicated expressions may be built to describe states of a system, and the transitions between them. They are often classified according to whether time is assumed to have a linear or a branching structure [CGP99]. For those two classes, one may refer to LTL (Linear Temporal Logic) [Pnu77] and CTL (Computational Tree Logic) [CES86].

The properties may also be encoded as *observers*. An observer is a transducer that outputs and *alarm* when a particular input sequence is recognized. In the context of embedded systems, most of the desired properties are safety or *bounded-liveness* properties, and may be effectively encoded into observers. Observers were intensively used in the synchronous approach [HLR93]. A components acting as an observer runs in parallel with the other components of the system. Because of the *synchronous broadcast* communication mechanism, the observer does not affect the functional behavior of the system.

2.4.2 Validation by Simulation

Validation by simulation consists in running the system to observe its behavior. It usually consists in providing the system with inputs and observing its outputs. The outputs are then compared to the expectation of the designer. Also, the simulation permits the observations of the state of the system under execution, for instance by choosing relevant break-points to observe some internal variables.

2.4.3 Runtime Verification

Runtime verification consist in checking some safety properties that the system must satisfy. Runtime verification requires the execution of the system. The properties are expressed in terms of logical properties, temporal logics, or automata (observers).

Usually, verifying properties at runtime requires the extraction of execution traces as a sequence of relevant events (function calls, assignment, etc.). The execution trace is given as input to a monitor that checks if the desired properties are indeed satisfied or not.

Several tools were designed for the purpose of runtime verification. Some of them allows for expressing various properties, possibly written in distinct logics (e.g., Java-MOP [CR05], j-VETO [Fal09], etc.). Others are dedicated to particular issues such as *race condition* detection in multi-threaded programs (e.g., FastTrack [FF09], Eraser [SBN+97], etc.), deadlock detection [JPSN09], etc.

Runtime verification requires intensive execution of the system in order to observe its potential behavior. As we cannot deal with infinite executions, one may say *the property has never been violated*, but we cannot state that *the property is never violated*.

2.4.4 Static Verification

Static verification analyzes a formal model of the system without executing it. The concrete semantics of a language may be used to define a formal model of a program written in that language. Such a model may expose infinite behaviors which makes most of the problems undecidable. In static verification techniques, the decidability problem is solved by using some form of abstractions.

The methods used for static verification differ in the way to perform abstractions. In the sequel we present *model-checking* and *abstract interpretation* techniques.

2.4.4.1 Model Checking

Model-checking [CGP99, QS82] allows for verifying finite state systems. This means that even if a system exposes an infinite number of states, its abstract model should be finite.

The abstract model is given manually by the user, or may be computed with techniques relevant to static analysis of programs. The abstract model is given as input to a model checker, together with the property to verify (expressed in temporal logic or as an observer). The model checker performs exhaustive exploration of the set of possible states. It is able to state whether a property is satisfied or not. In case it is not satisfied, the model checker provides a counterexample that violates the property.

Model-checking faces the *state explosion* problem. Elaborate techniques try to minimize the state space of the abstract models. For instance, *symbolic model-checking* consists in representing sets of states by formulas, and to encode formulas by BBDs (*Binary Decision Diagrams*) as

in the SMV model-checker [McM92b]. Other approaches apply *partial order reduction* in order to reduce the set of states to be searched (e.g., SPIN [Hol97, HP96]).

2.4.4.2 Abstract Interpretation

Abstract interpretation takes a program in its original form (i.e., not a model of it) and tries to compute its possible behaviors. However, the formal representation of a program as it is given by its concrete semantics may expose infinite behaviors.

Abstract interpretation [CC77, CC76] aims at associating an *abstract semantics* with the program that over-approximates its *concrete semantics*. Over-approximation means representing the set of possible behaviors by an abstraction containing at least these behaviors. Abstract interpretation associates *abstract* values (e.g., intervals, signs, etc.) with the variables of a program instead of their concrete values (e.g., numerical values). Interpreting the program through its abstract semantics computes an over-approximation of the possible values of the variables in terms of their abstract representation.

Abstract interpretation is applied to static analysis of programs [CC77, HP08], program optimization [CGS94], typing [Cou97], etc. A lot of tools were designed for the verification of programs by abstract interpretation such as ASTRÉE [BCC+03], TVLA [LAS00], Polyspace [plo], SLAM [BR02], etc.

Chapter 3

Overview of the 42 Model

Introduction (En) *This chapter presents an overview of the thesis. The basic elements of the 42 component model are introduced and illustrated by an example. We describe 42 components and how they are assembled in order to form other components. Components are viewed as black boxes which impose some kind of specification. 42 components may be described by means of explicit or implicit specifications. We study the difference between the two and describe a mixed type of specification. The 42 model is not meant to be a new language for the design of embedded systems. Instead, it is a tool for reasoning on components from various domains. The benefits of using 42 as a modeling tool are listed in this chapter. We present a tool developed for 42 to enable execution and simulation of components.*

Contents

Introduction (Fr) *Ce chapitre donne un aperçu du contenu de la thèse. A travers d'un petit exemple, nous décrivons la notion de composant 42, et comment assembler des composants pour en créer des nouveaux. Les composants 42 peuvent être décrits par des spécifications implicites ou explicites. Nous présenterons ces deux types de spécifications, ainsi qu'une forme de spécification mixte. 42 n'est pas un nouveau langage pour la conception des systèmes embarqués ; c'est un outil pour la réflection sur la notion de composants dans le domaine des systèmes embarqués. Nous montrerons l'intérêt d'utiliser 42 comme un outil de modélisation. Enfin, nous présenterons un outil qui à été développé autour de 42 qui permet d'exécuter des assemblages de composants 42 pour observer leur comportement.*

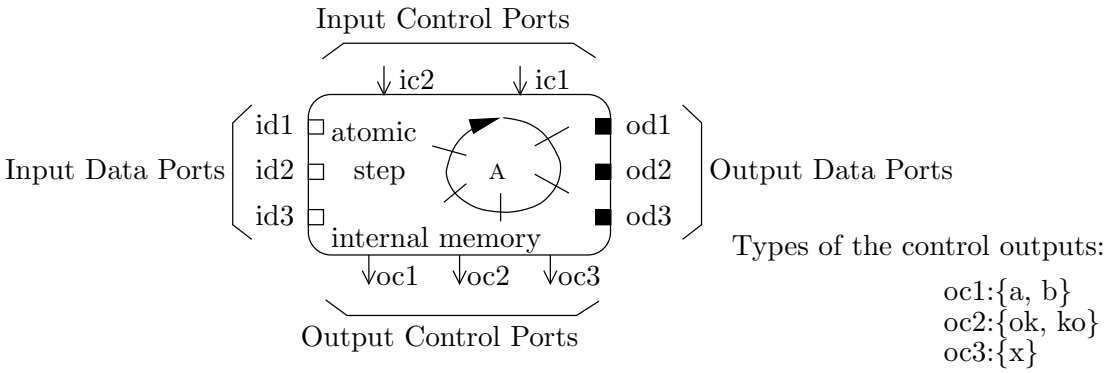


Fig. 3.1: External view of the 42 component A

3.1 Basic Elements of the 42 Model

42 is a component-based model designed for formal modeling of heterogeneous embedded systems. The basic elements of the model will be presented in this section: the notion of components, their interface, how to assemble components, etc. 42 is given as a formal specification (see Chapter 5) and is not associated with a particular language. For simulation purposes, one may use any language to implement a tool and any language to program components.

3.1.1 Basic Components

Figure 3.1 shows a 42 component named A. A component is a black box that has *input and output data ports*, and *input and output control ports*. The input control ports are used to ask it to perform one finite-execution. Since there may be several input control ports, there may be several *entry-points* that toggle an execution. An execution corresponds to a terminating (non-necessarily deterministic) piece of code. Allowing non-deterministic components means that 42 allows to mix code with specifications. A component may have some internal memory. The input and output data ports are used to communicate data between the components. The output control ports will be used by the components to send information to the controller (see below).

3.1.1.1 Components Interface

The set of input/output control and data ports constitutes the *interface* of a component. The interface is the only visible part of a component. The ports of a component are typed. Input control ports are used to activate it (they are of Boolean type). The type of the data ports may range from simple Boolean to complicated data-structures (e.g., arrays of integers or records, ...). The output control ports are used to communicate with the controller. They are of some enumerated type. On the right of Figure 3.1, the types of the output control ports are indicated. For example, `oc1` may take its values in the set `{a, b}`.

3.1.1.2 Implementation of Basic Components

Besides its interface, a component has an *implementation* that defines its behavior. Basic components are written in some programming language. Figure 3.2 is an example code for the component in Figure 3.1, written in some imperative style. For each control input, the component executes a program that corresponds to its activation. It should terminate in bounded time. The memory (all the global variables: `m, v`) is initialized when the component is instantiated somewhere, it is persistent across the successive activations of the component.

```
Component A (
  control input ic1, ic2: bool;
  control output oc1:{a,b}; oc2:{ok,ko}; oc3:{x}
  data input id1, id2, id3:int; data output od1, od2:int; od3:bool;)
  var m : int := some_init_value ;
  v: int;

    for ic1 do : {                          for ic2 do : {
      int cpt := id1.read();                  if(id2.hasValue())
      if(m < 0) m := - m;                     ...
      while (m > 0){                          if (m ...) {
        m := m - cpt;                           ...
        cpt ++;                                 v := id1.read();
      }                                         oc3.write(x);
      od1.write(cpt);                         }else{
      if (cpt>42){                              m := v * 42;
        oc1.write(a);                           ...
      }else{                                    if(...)
        oc1.write(b);                             ...
      }                                           od3.write(true);
      m := m + cpt ;                              ...
    }                                     }  }}
```

Fig. 3.2: Internal view of the component A in Figure 3.1

The ports of a component are unidirectional. That is, a component cannot write on an input port, and cannot read from an output port. The way components access the ports depends on the implemented tool. For instance, in the code of Figure 3.2, `id1.read()` allows for reading the port `id1`; `od1.write(cpt)` assigns the value of the variable `cpt` to the output port `od1`. We also allow components to test which of the inputs are made available. For instance, `id2.hasValue()` checks whether a value is available on the input `id2`.

42 does not impose a particular language for the individual components. When importing existing components (for instance the C code produced by the compiler of a synchronous language), they have to be wrapped so that the input control ports correspond to the activation of a piece of code (methods in an object-oriented language, functions in C, ...), and the data ports correspond to some of the parameters of the activations. For hardware components, the 42 data ports usually correspond to wires of the real hardware component.

Figure 3.3 for instance, illustrates the correspondence between a piece of C code and a 42 component. The control inputs of the component correspond to the functions in the code; the input/output data ports correspond to the parameters of the functions. Global variables (e.g., `m`) correspond to the internal memory of the component.

3.1.1.3 Partial Access to the Ports of the Interface

The activation of a component corresponds to one of its computation steps. It has the general effect of reading inputs, updating the internal memory, computing control and data outputs. However, as illustrated by the example code above (Figure 3.2), an activation may not use all the inputs, nor compute all the outputs. For example, the activation with `ic1` uses only the input `id1`, updates the global variables, computes the outputs `od1` and `oc1`. Moreover, two activations with the same input control port may access distinct ports of the interface.

When a component has such a complex behavior, there is often a need for a precise description of it. 42 components may be associated with explicit specifications (see Section 3.2.2) to expose relevant information about its behavior.

```
#include ...
...
int m=0;
void fun1(const int i, int* o){
  *o = (m + i)*42;
  ...
  m ++;
}

void fun2(int* o){
  *o = m * 42;
  ...
}
...
```

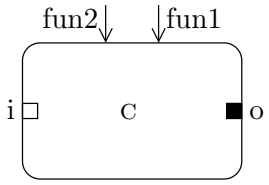


Fig. 3.3: Wrapping a piece of C code into a 42 component

3.1.2 Composed Components

***Components are assembled to form the* architecture *of a* system;** they are connected by point-to-point directed *wires* (see Figure 3.4). A wire relates two data ports of the same type (we will assume this is always true in the sequel). A system may have *global* data ports, in which case it is considered as an *open system*. Otherwise, in the absence of global data ports it is a *closed system*.

The wires are used to describe the architecture of the system. A wire may relate an output data port of a component to the input data port of another (potentially the same); a global input data port to the input data port of a component; an output data port of a component to a global output data port of the system. Wires do not mean a priori any synchronization, nor memorization. They indicate that some data may flow from a data port to another.

A *system* made of components connected by wires has no semantics until it is equipped with a *controller* (the box labeled with `ctrl` in Figure 3.4) to which the control ports of the components are connected implicitly. The controller activates the components, reads their control outputs and decides what happens on the wires.

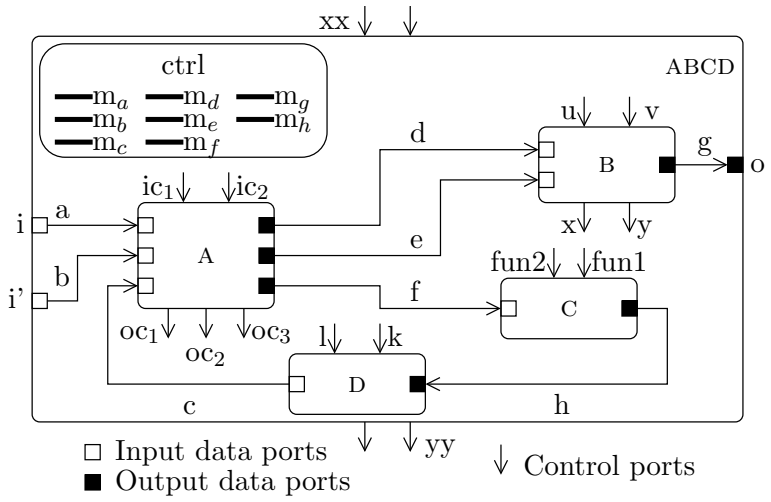


Fig. 3.4: Composing components and the ADL

A* system *is encapsulated to form a component. The interface of the *composed component* exposes new input and output control ports, and new input and output data ports (the global data ports of the system).

Figure 3.4 is an example of a composed component: the component ABCD is made of the set of components A,B,C, and D, the set of wires, and the controller `ctrl` to activate them. The encapsulation of components into a new one, hides the details of the implementation of the new component, and exposes its interface. Hence, a *composed* component is not distinguishable from a *basic* one.

The components used to design a new component may be *basic components* as it is the case for the components A and C. A (resp., C) is an instance of the component in Figure 3.1 (resp., Figure 3.3). Only the interface of the components B and D is known. They may be implemented as an assembly of other components. *Composed components* are also used to design new components which make the model *hierarchic*.

***The semantics of an assembly is defined by a* controller.** The controller defines how components behave together, and the resulting behavior of the composed component. Hence, the *controller* implements the *MoCC* (*Model of Computation and Communication*) associated with the assembly.

The controller is in charge of *translating* an activation request on one input control port of the composed component (e.g., `ABCD.xx`, also referred to as a *macro-step* in the sequel), into a sequence of activations of the subcomponents, and data exchanges between them (also called *micro-steps*); it also reports on the activity of the subcomponents, through global output control ports like `ABCD.yy`. To manage the communication between components, the controller may use some temporary variables explicitly associated with the wires.

The memory associated with the wires serves only as temporary storage, to build a macro-step (i.e., the activation of the component ABCD with `xx`). Its lifetime is limited to the macro-step. If there is a need for storing information (produced by a component) between two macro-steps, we can use the memory associated with the ports of the components (see Section 3.1.3 for a discussion on the memory). However, when the memory plays an important role in the system, or may have complex behavior, then there should be an explicit component behaving as a memory (see examples in Section 4.2.2).

Figure 3.5 is an example code for the controller `ctrl` written in a simple imperative style. It is used within the assembly of Figure 3.4 and implements a fictive *MoCC*.

In this example, within the activation with `xx`, the controller `ctrl` associates a bounded FIFO with each wire. The wires `a, b, c` of Figure 3.4 are associated with the one-place `int` FIFOs $\mathtt{m}_a$, $\mathtt{m}_b$, $\mathtt{m}_c$ of Figure. 3.5, the wires `d, e, m` are associated with the 4-places `int` FIFOs $\mathtt{m}_d$, $\mathtt{m}_e$, $\mathtt{m}_m$, and the wires `f, h` are associated with the 2-places Boolean FIFOs $\mathtt{m}_f$, $\mathtt{m}_h$.

A FIFO `M` offers three methods: `M.get` gets a value in `M` and puts it into the consumer port connected to the wire; `M.put` gets a value in the producer port connected to the wire and puts it into `M`; `M.init` initializes `M` to an empty FIFO. It is the responsibility of the controller to avoid writing in a FIFO when it is full, or reading from an empty FIFO.

In response to the global activation `xx` of the global component ABCD, the controller of Figure 3.5 executes the micro-steps defined by the program associated with the global control input `xx`. First, the controller initializes the memory associated with the wires. Depending on the value of the variable `m`, it may copy the value of the global input port `i` into the memory associated with the wire `a` with $\mathtt{m}_a$`.put`. The micro-step $\mathtt{m}_a$`.get` moves the content of the wire to the input of the component A. The controller activates the components A and D through their control

```
Controller ctrl is
var m : bool := true ;
for xx do {
  m_a , m_b , m_c : fifo(1,int);
  m_d , m_e , m_m : fifo(4,int);
  m_f , m_h : fifo(2,bool);
  if (m) {
     m_a.put; m_a.get;// reads i
     A.ic_1; // activates A
     D.l;  m_c.put; m_c.get;
     m_b.get; m_b.get; A.ic_2;
     m_d.put; m_d.get;
     m_e.put; m_e.get; B.u;
     m_g.put; m_g.get;// defines o
     m := B.x;
     m_f.put; A.ic_2; m_f.put;
     m_f.get; C.fun1; m_f.get; C.fun1;
  } else { ... }
  yy := true ; // defines yy
}....
```

Fig. 3.5: Code for the controller in Figure 3.4

input ports with `A.ic1, D.l` respectively. The value produced by the component D is copied from the output port of D to the input port of A with $\texttt{m}_c$`.put;` $\texttt{m}_c$`.get`, etc.

The program of the controller may also copy the control outputs of the individual components, into some memory local to the controller (e.g., `m:=B.x`), and whose life span may exceed the reaction to `xx` (inter-macro-step memorization). Finally, it may set a value for the global output ports (e.g., `yy := true`).

In the example code, the controller executes `D.l` without providing it with an input. This is because a component does not necessarily need all of its inputs (resp. produce all of its outputs) at all times (see below).

The memory $\texttt{m}_f$ receives 2 values before they are consumed by C. For a given *macro-step*, there is no general rule for the components to activate. A component may be activated twice as it is the case for the component C or never activated during the macro-step. This depends on the *MoCC* implemented by the controller.

3.1.2.1 From Data to Control and Vice-Versa

The restrictive constraints on the connections for the data and control ports are meant to enforce the identification of such a data/control classification for the components. Notice that, if the controller has to take decisions that depend on a data output `od` of some component C, the `od` port can be connected to the input data port $\texttt{d}_{in}$ of some special component COND (Figure 3.6-(a)) that produces a control output for the controller. For instance, such components are used for the modeling of multi-cycle programs in Section 4.2.2.3.

Similarly, if a global data output is in fact produced by the controller (and so cannot be connected to a data output of some component), we can just add a special component GEN (Figure 3.6-(b)) activated by the controller, and producing the desired data output.

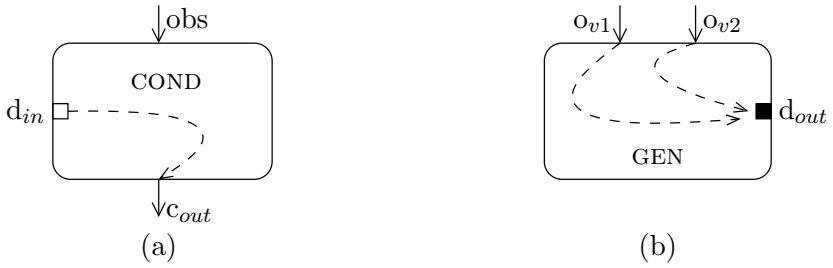


Fig. 3.6: From data to control values (a) and vice-versa (b)

3.1.3 Discussion on the Memory Associated with the 42 Model

The description of the 42 model involves some *memory* associated with the various elements of 42. Part of the memory may be actually associated with the final system and needs to be modeled explicitly, the rest of it is used for the purpose of modeling and simulation.

This section is meant to describe the various classes of memories in terms of their type, location, persistence, scope, etc.

3.1.3.1 Basic Components may Have Memory

The memory corresponds to the set of the global variables of the component. This memory may be shared between the various activations of the component. The global variables `m` and `v` of the component A (Figure 3.2) correspond to its memory. The memory of the component is not used for modeling purpose, it is indeed part of the system being modeled.

3.1.3.2 The Controllers may Have Memory

The memory corresponds to the set of global variables of the controller. It may be used by the controller to store the control outputs of the components as it is the case for the variable `m` of the controller `ctrl` of Figure 3.5. Also, the controller may use this memory to remember some choices it did during the previous activations, and to decide on the next activations of components. An example of such usage of memory will be illustrated in Section 6.2.2.

3.1.3.3 The Memory Associated with the Wires is Non-persistent

Such a memory is used for modeling purposes only, it is in fact part of the memory of the controller. It is a FIFO whose size is defined by the controller. At the beginning of each macro-step, the memory of the wires is initialized to empty FIFOs, losing any information from the previous macro-step. Hence, it is non-persistent.

3.1.3.4 Modeling Requires Persistent Memory

The various *MoCCs* we studied with 42 (see Chapter 4) demonstrated that there is indeed a need for some persistent memory in addition to the memory that is already present in the component themselves, and in the controller. Such a memory would be used by the controller to manage the communication between components; it is required for the purpose of modeling.

For instance, synchronous circuits do not require a shared memory to communicate, because components are alive at the same time. But, for the purpose of modeling synchrony, we need some implicit and persistent memory. To implement a synchronous *MoCC*, the controller should be able to activate components at least twice with the same inputs. These inputs have to

be stored somewhere. The modeling examples of synchronous systems (see Sections 4.1.1, and 4.2.1) give more details.

On the opposite, asynchronous components require explicit memory in order to communicate. There are two choices: either we model the required memory explicitly as a component, or we can abstract it; instead we would use the implicit memory.

Associating additional memory with the 42 model may be done in several manners. Our point of view is that:

- Associating the memory with the controller implies repetition, this would reproduce the architecture of the system.
- The additional memory cannot be associated with the components or the wires, because this would be meaningless w.r.t. the system being modeled. Moreover, in case a value is stored inside the component, the controller needs to activate the component each time the value is required.
- A better choice is to associate such a memory with the data ports of the components. The controller may access to the memorized value whenever it is required. However, it is not useful to have the persistent memory associated with the inputs and outputs together; one of them is enough.

3.1.3.5 Output Data Ports are Associated with Persistent Memory

For the rest of the thesis, we associate persistent memory with the output ports of the components. The lifetime of the memory associated with the input ports is limited. We will insist on the use of the memory associated with the inputs and outputs together with the description of the examples. Briefly, associating the implicit memory with the outputs imposes the following:

- If an input data port i is required by a component, the controller must put a value on that data port before activating the component to perform a computation step. During its computation step, the component may use i as a read-only temporary variable. At the end of the computation step, the value of i is lost.
- When a component is activated during a macro-step, it may put some values on some of its outputs. These values may be used by the controller to provide other components with their inputs:
  - during the same macro-step when component communication is synchronous (e.g., the examples of Sections 4.1.1, 4.1.2, 4.1.4, etc.),
  - during another macro-step when the communication is asynchronous (e.g., Section 6.2.2).
  - more than once, as it is the case for the synchronous controllers (e.g., Section 4.1.1, Section 4.2.2, etc.)

3.2 Specifying Components

In the context of 42, the example described above has shown that there is some need for a precise specification of the components, in particular for declaring which of the data inputs are needed for each control input, and which of the data and control outputs are produced. For a designer to be able to use components correctly (i.e., writing a correct controller), this information is crucial.

If the components expose simple behaviors, one may rely on *implicit* specifications (see Section 3.2.1); i.e., a way to associate an abstract description with a class of components. But if the behavior of components is complex and we need to express fine details about individual

components, we have to associate an *explicit* specification (see Section 3.2.2) with each of them. The designer of a component is in charge of providing such a specification. Also, we can consider components embedding their specifications (see Section 3.2.3). The component should be interrogated to know about its behavior. This supposes that the components are designed so as to provide introspection mechanisms.

3.2.1 Implicit Specifications

For some *MoCCs*, there is a possibility to describe components (or classes of components) that behave in the same way. When using a component from a specific class, we know how it should be used within an assembly of other components. Once the assembly is encapsulated in a new component, the designer makes the choice of the class to which the new component would belong. Typically, the examples in Section 4.1 are based on implicit specifications.

An example of implicit declaration is to say that the activation of any component requires all inputs and provides all outputs (one class of behaviors). We then know when a component may be activated, and when data is made available on its outputs. The general modeling of synchronous programs described in Section 4.1.1 uses implicit specifications, as well as the modeling of Kahn Process Networks in Section 4.1.4. We will discuss the limitations of implicit specifications together with these examples.

3.2.2 Explicit Specifications: Rich Control Contracts for 42

Implicit specifications are not general enough. In 42 we want to express fine details on the behavior of each component. As described by the example in Section 3.1.1, a component activation may not require all the inputs, nor provide all the outputs. Moreover, the data dependencies may be distinct between two activations with the same input control port.

With 42 we adopt the notion of *control contracts* as opposed to *data contracts* for this purpose. By data contracts, we mean some assumptions on the values of the inputs and outputs of a component. Basically, control contracts describe how the component should be activated, and what the required inputs/provided outputs are for each activation. Some of the values of the data ports may be specified because they are involved in the control, but in general, the contract expresses their availability only.

Figure 3.7 shows a contract for the component A of Figure 3.1. Each transition has a label of the form: `[condition] {data req} control input / control outputs {data prod}`

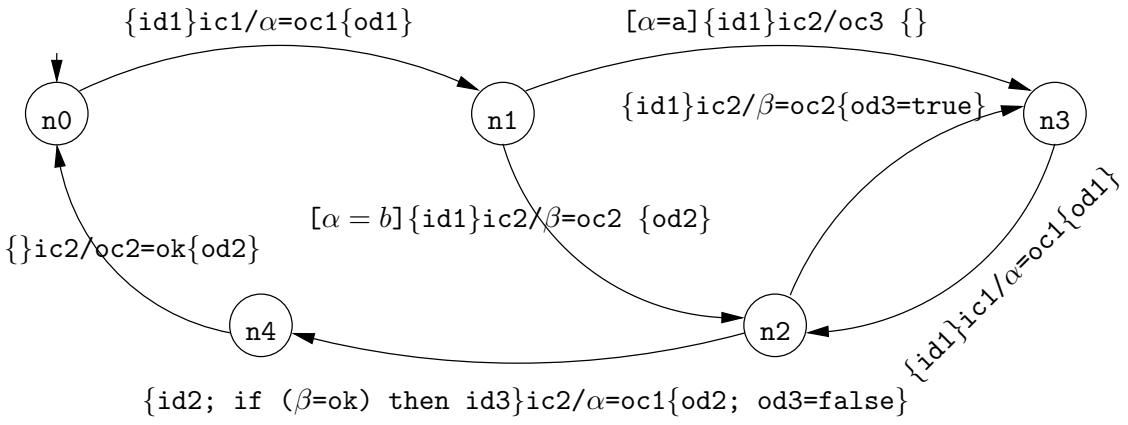


Fig. 3.7: Example contract for the component in Figure 3.1

The [condition] part of the label describes some conditions on the activation of the component with the input control port designated by the control input part. The {data req} part describes the required inputs, {data prod} describes the provided output data ports, and the control outputs part describes the provided control outputs.

We give more details on 42 control contracts in the sequel. Modeling examples with explicit contracts will be seen in Section 4.2. Section 5.2 formalizes the notion of 42 control contracts.

3.2.2.1 Sequence of Correct Activations of the Component

The structure of 42 control contracts is a finite-state automaton. This structure is used to specify the correct activation sequences of the component. Figure 3.8 illustrates an automaton describing the correct activations of the component A of Figure 3.1. The automaton is extracted from the contract of Figure 3.7 where we removed all details in the labels but the control input part.

Initially, the contract is in its initial state (n0), and then it evolves according to the sequence of activations produced by the controller in response to a sequence of macro-steps. Each macro-step is considered to start in the state where the contract was, at the end of the previous macro-step.

The outgoing transitions from a state correspond to the possible activations of the component. At a given state, the activation of the component with a control input not specified by the automaton is a violation of the contract. For instance, the component A should not be activated with ic1 at state n4.

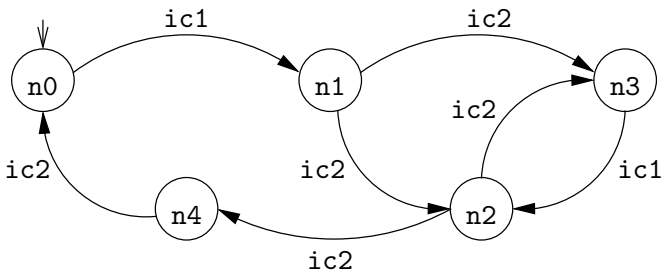


Fig. 3.8: Specifying the correct activation sequences

3.2.2.2 Data Dependencies

The activation of a component has the general effect of reading inputs, modifying the internal state, and producing outputs. For a given activation, not all inputs must be used nor all outputs must be produced. 42 contracts allow for a clear description of which inputs are needed for a given activation and which of the outputs will be given a value. When we look at the implementation of the component in Figure 3.2, we see that each time the component is activated with ic1, it requires id1 to provide od1. This information appears in the contract in the form of the following expression {id1}ic1/...{od1}.

The same control input may have distinct data dependencies depending on the state of the contract. This is the case for the input control port ic2. In Figure 3.7, the transition from state n1 to n3 (i.e., ...{id1}ic2/...{}) requires id1 and provides no outputs, while the transition n4 to n0 (i.e., ...{}ic2/...{id2}) requires no inputs and provides id2.

During an activation, a component is expected to read no more than the inputs declared by the transition of its contract. As the memory of the input ports is not persistent (see Section 3.1.3), the controller is responsible for providing the inputs to the component before the activation. On

the other hand, the component guarantees to the controller that at least the output declared will be overwritten. This means that the controller has no guarantees that some outputs will be unchanged.

Associating Values with Data Ports Data dependencies are expressed in terms of requirements/productions. In the general case, data dependencies deal with the availability of inputs (resp., the produced outputs) not with their assigned values. However, the value carried by some of the inputs/outputs may affect the behavior of a component. For instance, an interrupt wire may trigger the execution of an interrupt handler. This depends only on the value carried through the wire (`high/low`) not on its availability (an interrupt wire always carries a value). Such values should appear in the control contract of a component.

42 contracts allow to explicitly declare the values required/produced for some of the data ports. However, we require these ports to be of an enumerated type. For example, the transition labeled with {`id1`}`ic2/`β`=oc2{od3=true}` in the contract of Figure 3.7 declares that the output data port `od3` is provided after the activation and is assigned the value `true`. Some examples where the values of some data ports are described in the contracts may be found in Sections 4.2.3 and 4.2.2.

3.2.2.3 Control Outputs

Control ports are used to communicate with the controller. When activated, a component may compute some of the output control ports that may be read by the controller. The moment (the activation) when a control port value is computed is declared in the contract. In the contract of Figure 3.7, the control output `oc1` is given a new value in {`a, b`}. It is computed each time the component is activated with `ic1`. Moreover, the contracts allow to explicitly declare the value assigned to the control output (e.g., {}`ic2/oc2=ok`{}).

3.2.2.4 Using Extra Variables to Enhance the Readability of Contracts

The variables denoted by Greek letters are used in the contracts in order to refer to the values of control outputs. The variables may be used later on in the contract itself to express conditional activations, and conditional data dependencies. Let us note $\mathcal{V}$ the set of such variables.

- **Referring to control outputs values:** The transition from `n0` to `n1` labeled with {`...`}`ic1/`α`=oc1{...}` expresses that after the activation of the component with `ic1`, the component produces a value on the control output `oc1`. The value produced is referred to by the variable α.
- **Expressing conditional activations** The `[condition]` part of a transition label is built from the variables in $\mathcal{V}$. For instance, the variable α is used to express the activation condition in the transition `[`α`=b]{...}ic2/...{...}` (from `n1` to `n2`). The component may take this transition if and only if the value referred to by α is equal to `b`.
- **Expressing conditional dependencies** The {`data req`} (resp., {`data prod`}) part may include conditional data dependencies; the conditions are built on $\mathcal{V}$. For instance, the variable β is used to express conditional data dependencies in the transition {`id2; if (`β`==ok) then id3}ic2/...{...}` (from `n2` to `n4`). The transition means that the activation requires `id2` and, if the value referred to by the variable β is equal to `ok`, then it also requires `id3`.

The variables from $\mathcal{V}$ do not add to the expressiveness of the control contracts. They are used to enhance their readability. It is possible to expand a contract (see Section 5.2.2) in order to

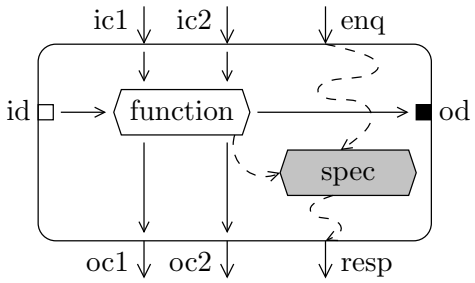


Fig. 3.9: A component embedding its specification

remove the variables from $\mathcal{V}$. The expansion yield an equivalent contract having a bigger set of states but the transition labels are simpler.

3.2.3 Components Introspection

We presented previously two kinds of specifications. Implicit specifications group components into classes, each class exposing a common behavior. Explicit specifications associate a control contract with each component. The contract describes precisely the behavior of the component. Here we introduce a kind of specification which may be considered as a mix of implicit and explicit specifications.

Instead of designing a component and providing its specification separately, one may decide to couple the functional behavior of a component with its specification. For each activation, the component knows the required inputs, and the provided outputs. It can be asked for its specification whenever it is needed. An example usage of introspection is described in Section 6.2.2.4. In this example, the introspection mechanism is used to compute, at runtime, the contract of a component depending on the contracts of its sub-components.

Typical Implementation we equip components with additional control inputs/outputs. The additional control ports are used for modeling purposes. They are not part of the functionality of the component. This resembles the introspection mechanism provided by some programming languages (e.g., Java, Objective-C, etc.) or some component-based models (e.g., *Fractal*). Introspection allows for getting some information from a component about itself.

Figure 3.9 illustrates a component in which the specification is part of the implementation. Its interface exposes the control ports `ic1, ic2, oc1, oc2` which are part of the functionality of a 42 component; and the control ports `enq` (for enquire) and `resp` (for response) which are used for introspection. As illustrated by the figure, the implementation of the component is split into two distinct parts (parts are not components).

The `function` ***part*** corresponds to the part of the implementation which is in charge of implementing the control inputs `ic1, ic2`, reading the data input `id`, computing the data output `od`, and updating the control outputs `oc1, oc2`. The `spec` part is in charge of implementing the control input `enq` and giving a value to the control output `resp`. When the component is activated with `enq`, it is supposed to provide some information through `resp` (i.e., possible activations, required data inputs, updated outputs). Notice that the `spec` part may need some information from the `function` part.

The `spec` ***part*** of the component may be seen as an implementation of a 42 control contract. The states of the contract correspond to the state of some internal variables of `spec`. Each

```
Controller intro is
...
for xx do {
  ...
  C.enq;
  let t=(req,ic,ocs,prod) ∈ C.resp;
  forall id ∈ req
      wire_id.get; //wire_id is connected to id
  C.ic;
  ...
}....
```

Fig. 3.10: Sketch of a controller using the component in Figure 3.9

time the component is activated with `enq`, `spec` computes a set of tuples. Each tuple is of the form `(req,ic,ocs,prod)` where `req` is the set of required input ports for the activation of the component with `ic`. `ocs`, and `prod` are the set of control outputs and data outputs that will be produced after the activation of the component with `ic`.

Typical Usage To use a component as the one of Figure 3.9, we know *implicitly* that it should be activated with `enq`. The component responds through its output control port `resp` with an *explicit* description of its possible activations. Hence, the kind of specifications we are dealing with is a mix of implicit and explicit specifications we described previously.

Figure 3.10 illustrates part of the code of a controller interrogating a component before activating it. First the controller ask the component for its possible activations: It activates the component with `enq`. The controller then reads the set of tuples provided by the component through `resp`. From this set, it selects only one tuple `t`. The set `req` associated with `t` contains the required inputs. For each input from this set, the controller moves a value from the wire connected to it. After the component has been provided with the required inputs, the controller activates it with the control input `ic` specified by the tuple `t`.

3.3 Consistency Issues

Mixing general controllers, rich control contracts, and the hierarchy of controllers, raises some consistency problems. The various notions of consistency between controllers/components and contracts are formalized in Section 5.3. Here we give an overview of such properties. In short, consistency answers the questions:

- **Is a basic component a correct implementation of a contract?** Typically a contract is an abstract description of the behavior of a component. If the component is used according to its contract, there should be no instance where it behaves unexpectedly, except if its implementation is wrong. The implementation of the component A (Figure 3.1) illustrated in Figure 3.2 must respect the contract illustrated in Figure 3.7. Respecting the contract means that for any possible transition in the contract, the activation of the component with the corresponding control input must not read an input data port not declared by the contract and must produce at least all the outputs declared by the contract.

  Section 5.3.1 formalizes in general the consistency of a (basic or composed) component implementation with its contract.

- **Is a controller using the component correctly?** 42 does not impose any particular language for the design of the controllers as long as they use the pre-defined primitives to activate components (i.e., the micro-steps). Moreover, any control structure provided by the language used to write the controllers is allowed. This renders the controllers very liberal, and allows them to do whatever the designer needs. Control contracts are used as safe-guards to prevent the controllers from any incorrect usage of components. In Figure 3.4, the controller `ctrl` must activate the component A with a sequence of activations recognized by the contract of A (Figure 3.7). Moreover, if an activation requires some inputs, the controller must provide the component with the required inputs. This kind of consistency will be formalized in Section 5.3.2.

- **Is a composed component a good implementation of a contract?** When a component is made as an assembly of other components (e.g., the component ABCD of Figure 3.4), the controller is in charge of defining the behavior associated with each control input. The controller is also in charge of reading the global input data ports (the micro-step `put` on the wire connected to that port) and producing the global control outputs and global output data ports (the micro-step `get` on the wire connected to that port). Hence, it is the responsibility of the controller to respect the contract of the composed component. For instance, if we associate a control contract with the component ABCD, the controller `ctrl` must not read a data input when it is not specified by the contract of ABCD, and must produce the values of the data and control outputs as declared by the contract.

  Formalizing the consistency of the controller (e.g., `ctrl`) with the contract of the encapsulating component (e.g., ABCD) amounts to formalize the consistency of the encapsulating component with its contract. It is presented in Section 5.3.1.

3.4 Using the 42 Modeling Approach

42 is essentially a tool for reasoning on components for *heterogeneous embedded systems*. We do not claim that it is a new language for the design of embedded systems. However, the design choices we made for 42 make it possible to use 42 jointly with existing approaches.

We list the benefits we can get when using 42. This gives an overview of what is described in the rest of the thesis. In the sequel, Section 3.4.1 deals with the benefits we gain when considering 42 as a modeling tool to describe the structure of a system. Section 3.4.2 describes how contracts may add to the expressiveness of the models, and to what extent they may be used in the modeling approach adopted for 42.

3.4.1 Reasoning on Components with the 42 Model

3.4.1.1 Describing the Interface of Components

42 may be used jointly with other component-based formalisms as a tool for reasoning on components. That is, given an entity from other approaches we try to define the corresponding 42 component. The 42-ization of components is a good exercise; it helps identify the parts associated with control and those associated with data. Moreover, applying the *FAMAPASAP* principle yields a rich description of the interface of a component. The interface contains the necessary details of the component to be used in an assembly.

In Section 7.3.1 for example, we describe first steps formalizing SystemC-TLM components with 42. We address the correspondence of SystemC-TLM components with 42 ones. Moreover, we will be able to import existing software or hardware components from SystemC, and to wrap

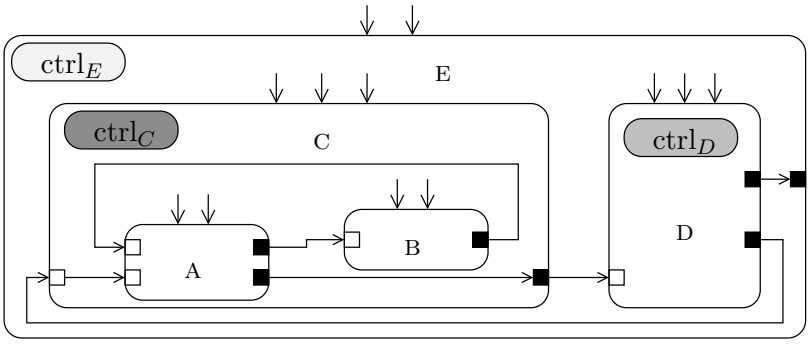


Fig. 3.11: Modeling heterogeneous systems with 42

them into 42 components.

The 42-ization of components exposes all the details about the control flow through the ports of the components. Sometimes the resulting interface contains a lot of details, as it is the case for the 42-ization of SystemC-TLM. However, 42 should not be blamed for that. The 42 interface exposes the required information about SystemC-TLM components when assembling them. This information is hidden in the simulation mechanics of SystemC, but the designer has to be aware of. The 42-ization of SystemC-TLM has at least the benefit of exposing all the control dependencies.

3.4.1.2 Expressing Various *MoCCs*

The behavior of an assembly is managed by the controller which describes various types of concurrency and structured interactions by means of basic primitives (i.e., the micro-steps). The controller organizes the micro-steps in a sequence according to the concurrency model described by the implemented *MoCC*. The power of the 42 controllers is their ability to describe several *MoCCs* with a small set of primitives.

Chapter 4 is a suite of examples for describing several *MoCCs*. The examples model systems ranging from pure synchrony (Sections 4.1.1 and 4.2.1) to pure asynchrony (Sections 4.1.2 and 4.2.3). They expose the interaction between components for each *MoCC*.

3.4.1.3 Modeling Heterogeneous Systems

Modeling heterogeneity requires the use of several *MoCCs* within the same system. Since 42 (as Ptolemy) does not allow for the use of several *MoCCs* at the same level, heterogeneity is dealt with by putting several controllers (implementing distinct *MoCCs*) in a hierarchy.

Figure 3.11 illustrates such a hierarchy of controllers. The components C and D encapsulate distinct *MoCCs* (i.e., `ctrl`$_C$ and `ctrl`$_D$). The controller of the component E (`ctrl`$_E$) knows how to make the components C and D evolve together. A *GALS* (Globally Asynchronous Locally Synchronous) system is a good example of an heterogeneous system. The modeling of such a system with 42 is described in Section 4.1.5.

3.4.2 Main Usage of 42 Control Contracts

3.4.2.1 Checking Consistency

Besides the fact that contracts constitute a rich description of a component, they may be used to check the consistency properties introduced in Section 3.3.

The consistency property between a component and its contract may be checked dynamically using the contracts. Whenever the component performs an execution step, we check whether it behaves as expected, i.e., if it didn't read more than the required inputs expressed by the contract, and did produce the expected outputs. Dynamic checking of such a consistency property is described technically in Section 6.1.1.

The consistency of the controller with the contract of a component was introduced in Section 3.3. The contract may be used as a monitor to check if the controller uses the component correctly. Whenever the controller is about to activate the component with a sequence of micro-steps not consistent with the contract, a consistency error is raised. Checking the controller consistency with the contracts is presented in Section 6.1.2.

The feasibility of a static check of consistency depends on the expressive power of the language used to implement components/controllers. In the context of 42, we favor expressiveness, not the possibility of static verification.

3.4.2.2 Automatic Generation of Controllers

Writing a controller to manage the execution of components may be a complicated task, especially when components have complex behaviors. This has to take into account the details given by each contract, and the architecture of the system to ensure that the resulting controller is consistent with the contracts of the components.

Instead of designing the controller and checking its correctness later, one can *generate* it. Automatic generation of controllers depends on the *MoCC* and guarantees the controller to be consistent with the contracts of components. It relies on the information provided by the data dependencies defined by the architecture, and the contract of each component.

Automatic generation of controllers is possible for some *MoCCs*. Sometimes, the architecture of the assembly, together with some implicit specifications of the components, is sufficient to deduce the controller; this applies to the mono-clock synchronous *MoCC* example of Section 4.1.1. For other *MoCCs*, automatic generation requires also the details provided by the contracts of the components (e.g., Section 6.2.2). The information provided by the contracts may be enforced with some explicit constraints, so that the resulting controller exposes some behaviors we are interested in (see the master/slave relation below).

The controllers may be generated *statically* or *dynamically*. Static generation yields a controller where the sequence of activations of components is known at compile-time (i.e., controllers as the one described in Figure 3.5). Section 6.2.1 describes such a type of automatic generation for the *synchronous MoCC*.

Controllers resulting from the second category compute the set of micro-steps associated with a macro-step at runtime. They need the contracts of the components to get the information on the state of the components and their possible activations. Section 6.2.2 describes such type of controllers which are based on contract interpretation.

When the components are equipped with introspection mechanisms, it is also possible to deduce the controller dynamically. Instead of relying on the explicit contracts, the controller should ask the components for their specifications at each macro-step. Hierarchic contract interpreters describe these controllers (see Section 6.2.2.4).

The Master/Slave Relation is an abstract means to express more constraints on the behavior of an assembly of components. It imposes the activation of some components to be in the same macro-step. It is therefore related to the atomicity of a macro-step. A master/slave relation relates a control output of a component to the control input of another. It may be used

to describe situations where a component requires the activation of another one (e.g., function calls).

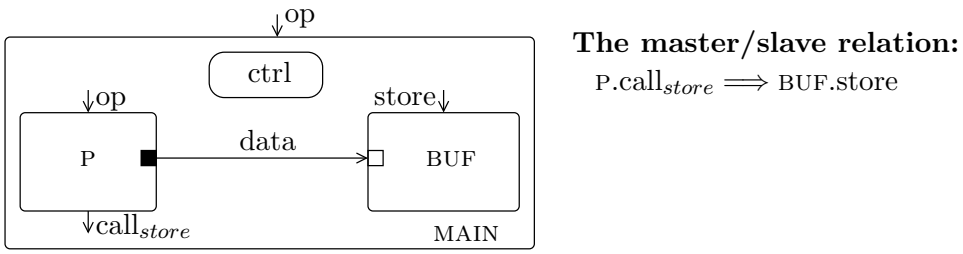


Fig. 3.12: The master/slave relation between components

Figure 3.12 illustrates a model of a producer (P) which requires its produced data to be stored immediately in the buffer (BUF). That is, each time P computes some value, BUF should be activated. The activation of components is the responsibility of the controller, because the control ports of the components are implicitly connected to the controller, not directly to each other.

The implication on the right of the figure describes the master/slave relation; it tells that: if the activation of the component P produces the control port `call`$_{store}$, the controller should activate the component BUF with its input control `store` within the same macro-step. We will see some examples of the use of the master/slave relation in Sections 6.2.2.7 and 7.3.1.

3.4.2.3 Executing Non-Deterministic Components

42 contracts are abstract and non-deterministic descriptions of the behavior of components. For some *MoCCs* a contract may be used as a possible and abstract implementation of a component; that is, the contracts are executable. Hence, if we require to execute a system early in the design flow, one may rely on the contracts describing the components. If we are happy with the execution of the contracts, we start writing the detailed implementation of the components. Contract execution will be presented in Section 6.2.2.

Section 7.3 is a complete example of reasoning about components in 42, and includes contracts-based simulation. It tackles the 42-ization of SystemC-TLM components. The contracts of these components may be extracted automatically from existing C++ code, or written by hand. We are able to execute such contracts, which yields a lightweight simulation model. With such an abstract view, one can reason on the synchronization of components without looking at the implementation details. Moreover, we are able to check some properties on assemblies of components.

3.5 Implementation

Based on the specification of 42, we developed a tool for experimentation purposes. The tool is written in Java. It is presented as a package and defines the classes required for describing the notions we have seen so far (i.e., components, ports, controllers, etc.). The tool includes features to instantiate components and execute them to allow for simulations.

3.5.0.4 Basic Components

Basic components are typically written as Java classes. The class implementing the behavior of a component should provide a set of methods. Each method corresponds to a control input of the component.

The code imported from other frameworks is wrapped in order to comply with the interface of 42 components. Sometimes this requires the use of interfacing frameworks between Java and the language of the imported code. In Section 7.2.1 we use *JNI* (Java Native Interface) to make C-code execute in a 42 component. Section 7.3.3.3 deals with wrapping C++ implementations of SystemC-TLM components into a 42 component.

3.5.0.5 Controllers

The controller `ctrl` (Figure 3.5) used with the example of this chapter is described with an imperative-style language. The tool enables the execution of the code of such controllers in order to expose the behavior of the assembly of components. Moreover, the tool may be extended easily to accept controllers written in distinct styles and programming languages. For instance, we introduced controllers acting as contract interpreters for the simulation of asynchronous systems. This kind of controllers will be described in Section 6.2.2.

3.5.0.6 Describing Components and Architectures

Each component is associated with two description files: an *interface description* file, and an *implementation description* file. Both of the description files are encoded into XML.

The interface description file describes the set of input/output control/data ports of the component, it will be describe in Section 9.1.1.

The implementation description file is used to describe the internals of the components. For a basic component, the implementation file refers to the Java class implementing the behavior of the component. For a composed component, the implementation file uses a simple *ADL* (Architecture Description Language) to described the architecture (see Section 9.1.2).

The tool includes a graphical editor to assemble components. It is based on *GMF* (Graphical Modeling Framework) provided by *Eclipse*. The graphical editor includes features to generate the *XML* files for describing the interface and the implementation of the components, and the Java code skeleton of basic components.

3.5.0.7 The Execution Engine

Once we define the architecture of a system, we can instantiate the components and start simulating the system. There is an XML parser that parses the architecture of the component at the top of the hierarchy. The parser goes in depth through the levels of hierarchy, to perform a bottom up instantiation; i.e., the tool starts instantiating basic components which are the Java classes at the leaves of the hierarchy, binds them with the wires, and instantiates the controller. This ends up the instantiation of the encapsulating component. The instantiation continues recursively until instantiating the component at the top.

At the time being, the implementation of 42 in Java is used as a prototype. It is not yet available for downloads, because of the incompleteness of the prototype. In particular, some work has to be done for enhancing debugging facilities. Moreover, the connexion between the graphical interface and the execution engine is not established.

Besides the experiments presented in the thesis, the tool serves for other experimentations. For instance, it is used to experiment the use of 42 to describe web-service architectures and BPEL (Business Process Execution Language) orchestration.

Chapter 4

Modeling Examples with 42 Components

Introduction (En) *This chapter is presented as a suite of modeling examples with 42. It deals with the modeling of several* MoCCs *and how are they combined to model heterogeneity of embedded systems. We illustrate some guidelines for designing components for some* MoCCs*, and how expressive the controllers are to implement various* MoCCs*. Within the examples, components are described with implicit or explicit specifications (i.e., the control contracts). We also demonstrate the expressiveness of control contracts for describing components.*

Contents

Introduction (Fr) *Dans ce chapitre, nous présentons une suite d'exemples de modélisation en 42. Nous nous intéressons à la modélisation de différents* MoCCs *en 42 et à leur combinaison pour modéliser l'hétérogénéité des systèmes embarqués. Nous décrivons aussi quelques directive pour la modélisation de composants pour certains* MoCCs*, ainsi que les programmes associés aux contrôleurs qui implémentent ces* MoCCs*. Les exemples sont regroupés en deux grande sections. Dans la première section, nous décrirons des composants associés à des spécifications implicites. Nous montrerons dans la deuxième section, l'expressivité des contrats de contrôle pour décrire des composants ayant un comportement complexe.*

4.1 Examples with Implicit Specifications

The purpose of this section is to provide some examples of modeling several *MoCCs* with 42. The examples focus on the basic notions of 42 described in Chapter 3. We consider components with implicit specifications only. Through the examples, we describe some guidelines for modeling components for some *MoCCs*, and how to write the corresponding controllers to implement the *MoCCs*.

4.1.1 Mono-Clock Synchronous Programs or Circuits

In a pure synchronous model of computation (for describing synchronous circuits for instance), the controller should be able to express the fact that, at each instant of a global clock, all the components of the circuit take their inputs, and compute their outputs (see Section 2.1.1 for more details). It may take some physical time to stabilize, but for non-cyclic circuits, it does stabilize. In the context of a component model like 42, with components and interactions between them, being able to express pure synchrony means that we should be able to describe the steps of the stabilization phase; it is not a simple interaction.

```
node DINTG (i: int)
returns (o: int) ;
var x, y : int ;
let
 x = INTG (i + (0->pre y)) ;
 y = INTG (x) ;
 o = y ;
tel.
node INTG(i : int)
returns (o : int) ;
let
 o = i -> pre(o) + i ;
tel.
```

Fig. 4.1: An Example synchronous program (in textual Lustre)

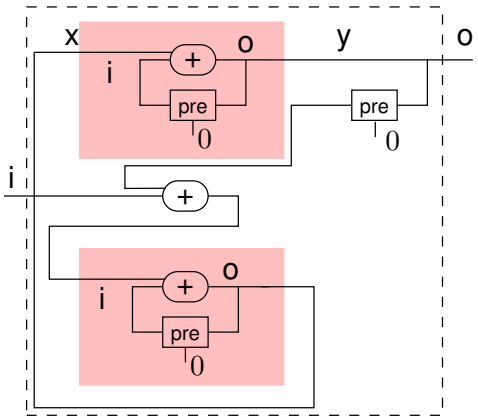


Fig. 4.2: The same program in a graphical form

4.1.1.1 Example

Figures 4.1, 4.2 and 4.3 illustrate the componentization of a Lustre program. We chose Lustre because its graphical form (as used in the commercial tool SCADE) is very close to the diagrammatic view used in synchronous hardware design. The program of Figure 4.1 is made of two instances of a basic integrator INTG. "`o = i -> pre(o) + i`" means: the output `o` is equal to the input `i` at the first instant, and then, forever, it is equal to `i` plus its previous value `pre(o)`. The two copies are connected in the node DINTG, with another addition operator. Figure 4.2 is a flat and graphical view of the node DINTG, where the two copies of INTG have been expanded.

4.1.1.2 Individual Components

Figure 4.3 is the component view of the program in 42. The level of the controller `sync1` is the level of DINTG. The components INTG1 and INTG2 are instances of the same component. The details of the instance INTG1 are hidden, INTG1 is considered as a basic component, as PLUS, PRE (a flip-flop, or elementary memory point) or the duplicator DUP. For illustration purposes, the instance INTG2 is described as a 42 assembly of more primitive components.

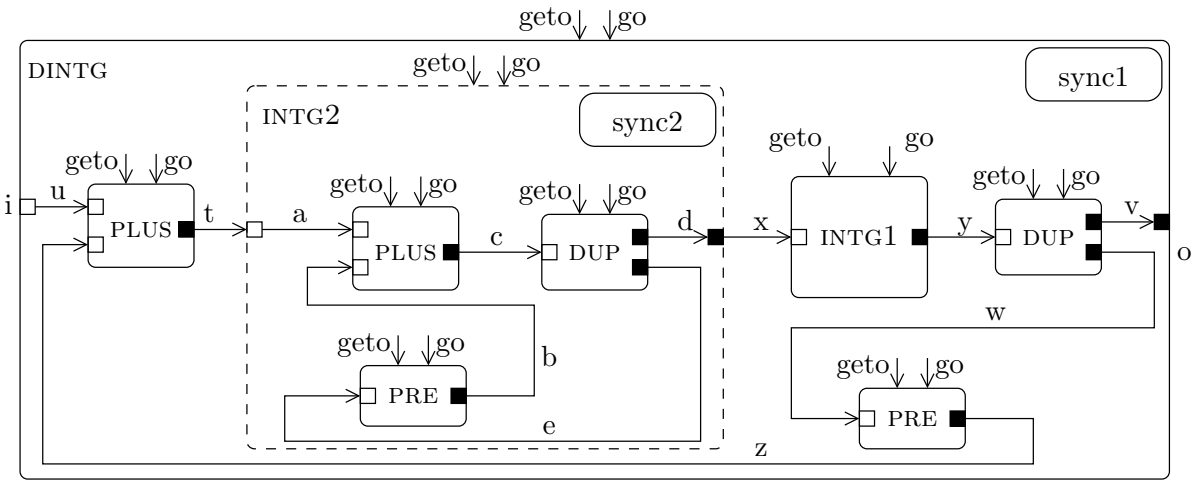


Fig. 4.3: A 42 component view of the program in Figure 4.1

```
Controller sync1 is {

  for geto do: {
    var u, v,..., z, t : fifo(1,int);
    PRE.geto; z.put; z.get;
    u.put; // reads i
    u.get; PLUS.geto;
    t.put; t.get; INTG 2.geto;
    x.put; x.get; INTG 1.geto;
    y.put; y.get; DUP.geto;
    v.put; v.get;  // defines o
  }

  for go do: {
    var u, v,..., z, t : fifo(1,int);
    w.put; w.get;
    PRE.go;
    x.put; x.get; INTG 1.go;
    y.put; y.get; DUP.go;
    t.put; t.get; INTG 2.go;
    u.put; u.get;
    z.put; z.get; PLUS.go;
        }}
```

Fig. 4.4: The programs of the controller `'sync1'`

```
Controller sync2 {

  for geto do: {
    var a, b, c, d, e : fifo(1,int);
    PRE.geto; b.put; b.get;
    a.put; // reads t
    a.get; PLUS.geto;
    c.put; c.get; DUP.geto;
    d.put; d.get;  // defines x
  }

  for go do: {
    var a, b, c, d, e : fifo(1,int);
    e.put; e.get;
    PRE.go;
    a.put; a.get;
    b.put; b.get; PLUS.go;
    e.put; e.get; DUP.go;
        }}
```

Fig. 4.5: The programs of the controller `'sync2'`

```
Component PRE (
control input geto,go : bool;
data input id:int;
data output od:int)

{
    var m : int := 0;
    for geto do :
    { od := m; }
    for go do :
    { m := id; }
}
```

Fig. 4.6: Code of the PRE component

```
initializations ;
while true {
  // read input
  int v = read();
  i.put (v) ;
  // write the value produced
  DINTG.geto;
  write(o.get) ;
  i.put (v) ;
  DINTG.go;
}
```

Fig. 4.7: Structure of a program using the main component DINTG

In order to obtain the normal behavior of a Lustre program (or a synchronous circuit) with such a component view, the components should be designed so that they offer two control inputs `geto` and `go`. When asked with `geto`, the component delivers its outputs, depending on the internal memory and its data inputs, but without changing internal state; `go` asks it to change its internal state. In this simple case, there is no need for control outputs. The combinational components (the PLUS, and the duplicator) have an empty `go` function.

4.1.1.3 Implicit Specification of Components

The black box view of components requires the components to be equipped with a specification. In this simple example we consider two classes of implicit specifications:

- A first class includes the components that require all their inputs to compute all their outputs. This class comprises the components like PLUS, DUP, INTG, etc. To update the internal state, the activation with `go` requires all the inputs. The components of this class are *Mealy-style* components.
- The second class describes *Moore-style* components; i.e., components in which the computation of the outputs depends on the internal memory of the component only. That is, `geto` requires no inputs and provides all the outputs. The component PRE in our example, is the unique element of such a class. Figure 4.6 is the code of the PRE component (for integers). Notice that the activation with `go` (to change internal state) requires all the inputs of the component.

Assembling synchronous components yields a synchronous component. One may wonder to which class of components this composed one should belong. For a composed component, the most general case is to consider it as a *Mealy-style* component, belonging to the first class of specifications.

4.1.1.4 The Controller

At the two levels of the hierarchy, the controllers associate one-place buffers with all wires. The programs they play when the global `geto` or `go` control inputs are activated are given in Figures 4.4 and 4.5, in some imperative style. At the beginning of each activation, the memory associated with the wires is initialized to empty FIFOs.

When a component (DINTG or INTG2) is activated with `geto`, the inputs are supposed to be available (this is implied by the implicit specification). The controller asks each subcomponent

to produce its outputs, according to the values that are available on its input ports. This is done in an order compatible with the partial order of data dependencies (see below).

For `sync1`, the only component that can start is the PRE component, because its output does not depend on its input. Then the PLUS can play, then INTG2, then INTG1, then the duplicator DUP. At the end of this sequence, all the wires have a value, the circuit has stabilized. The values cannot be stored on the wires, but they are persistent in the output ports (see Section 3.1.3 for the discussion about the memory associated with the output ports).

When the component DINTG is activated with `go`, the controller `sync1` asks each subcomponent to change state. The controller provides the inputs for each component before it activates it. The `go` activations of all the subcomponents can be called in any order.

4.1.1.5 Consistency of the Controller with the Data Dependencies

During a clock cycle, the controller should be able to describe the stabilization phase of the circuit. This corresponds to the situation where all components compute their outputs depending on the inputs they just received. That is, the sequence of activations with `geto`. Components must be activated in an order consistent with the partial order of data dependencies.

In the simple case we presented above, the architecture (the point-to-point directed wires) is sufficient to describe such a partial order, because of the implicit specification associated with the components. We know that all components require all their inputs to compute their outputs (except the PRE components). The partial order of the activations of INTG2 subcomponents is describe by the following:

$$a.put \prec a.get \prec \text{PLUS}.geto \prec c.put \prec c.get \prec \text{DUP}.geto \prec e.put \prec e.get$$
$$\text{PRE}.geto \prec b.put \prec b.get \prec \text{PLUS}.geto$$

This partial order must be respected by the controller `sync2` when the component INTG2 is activated with `geto`. To construct such a partial order, the rules are simple and are implied by the architecture and the specifications:

- **From the architecture:** the operation `put` on a wire always precedes the `get` operation. Because if the FIFO associated with a wire is empty the controller may not deliver the input for the component connected to this wire.

- **From the specification:** the activation of the first class components (i.e., *Mealy-Style* components) with `geto` requires all the inputs and provides all the outputs. Thus, the operation `put` on the wires connected to the data inputs of a component must precede the activation of the component with `geto`. Symmetrically, the operation `put` on a wire connected to an output data port must be after the activation of the component with `geto`. For instance, `a.get` and `b.get` both precede PLUS`.geto` which in turn precedes `c.put`.

  The activation of the components of the second class (the PRE components) with `geto` does not require inputs, thus it has no dependencies (e.g., PRE`.geto`). However, it must precede the operation `put` on the wire connected to the output of the component (e.g., `b.put`).

  The operation `put` on a wire connected to a global input data port has also no dependencies (e.g., `a.put`). This is because in synchronous modeling with implicit specifications, a composed component belongs to the first class of specifications. The global inputs are supposed to be provided before the activation with `geto`.

4.1.1.6 Comments

Modeling synchrony requires additional memory. During a clock cycle, the inputs used (by `geto`) to compute the outputs and those used (by `go`) to change state must be the same. This requirement is not expressed by the specifications, but it is related to the synchronous *MoCC*.

Using the same inputs for distinct activations imposes the memorization of the inputs provided to the component. This is made possible thanks to the memory associated with the output ports of the components (see Section 3.1.3). The controller `sync1` for instance, would not be able to provide the component INTG2 with the same inputs if the memory of the output port of PLUS was not persistent.

The main program requires memory. The code of a main program using the component DINTG is given in Figure 4.7. The function `read()` may be associated with a primitive function getting some value from a sensor. The value has to be stored in some variable (e.g., `v`), so that the main program may provide the same inputs during the activation with `geto` and `go`. Storing the value is necessary because the memory associated with the input port of DINTG is not persistent (see Section 3.1.3), and a second `read()` from the sensor may provide a distinct value.

The controller must be consistent with the specifications. A sequence of *micro-steps* associated with a control input of a controller must be consistent with the partial order of data dependencies. For instance, the `geto` control input as implemented by the controller `sync2` defines an order of *micro-steps* consistent with the partial order presented above

Notice that for a partially ordered set, there exists, potentially, several total orders consistent with it. The partial order may be used to generate the controller code. It amounts to computing a total order consistent with the partial order (using topological sorting algorithms [Kah62]). Automatic generation of controller code is presented in Section 6.2.1.

The example may be extended. One may describe conditional or partial dependencies between outputs and inputs (see below), and also multi-cycle synchronous programs (in which a subprogram should be run at speed s1, and another at speed s2 much slower than s1). The principle of the component view can be used for separate compilation of Lustre/SCADE programs, with some optimizations in memory management. All of these extensions require accurate specifications associated with each component.

4.1.1.7 Runtime Verification by Means of Observers

The synchronous example we described in 42 is executable. One can use some *observers* (see Section 2.4.1) in order to perform *runtime verification* (see Section 2.4.3). The observers are encoded into 42 components, and are connected to the outputs of the components to be checked. Adding observers does not alter the functional behavior of synchronous systems because of the *synchronous broadcast* communication mechanism.

4.1.1.8 Limitations of the Implicit Specification in the Synchronous *MoCC*

We describe in the sequel, some issues with the design of synchronous systems where the implicit specifications do not give adequate solutions.

Partial Computation of the Outputs. In general, components have more than one input and one output. There are two choices: either we consider that all the outputs depend on all the inputs, and in this case we can apply the previous scheme. Or we can accept more complex designs, in which an output does not necessarily depend on all the inputs. In this

case, each component has to specify the dependency between its outputs and its inputs, and each component has to be equipped with a `go` activation, and one `geto` activation per data output.

If we require such expressiveness, implicit specifications are no longer usable. Section 4.2.1 deals with the modeling of synchronous systems with explicit specifications. Each component is equipped with its control contract. If the specifications are expressive enough, the controller can then ask the components to produce specific outputs, not all at a time, and interleave the computations of the outputs of the subcomponents.

Exposing Potential Moore-style Components. The 42 component view of a synchronous program requires that there is indeed a possible computation order. Cycles in the data dependencies do not allow for the computation of partial orders. Thus, we require that each cycle in the data dependency graph be cut by a Moore-style component (e.g., a component PRE).

Within a synchronous system, composed components may be expanded yielding a system composed of basic components. This allows to observe all the possible cycles and if they are indeed cut by a PRE Moore-style component. The encapsulation of components into a composed one has the effect of hiding potential Moore-style components that may break the cycles. Implicit specifications are not expressive enough to expose that a cycle is indeed broken inside a component, we should use explicit specifications to gain such an expressiveness.

4.1.1.9 Consequences of the Limitations

Consider for example the synchronous model in Figure 4.8. The component ABC is composed of two components: AB and C. These two components require all their inputs to produce all their outputs. At the level of ABC, the component AB is considered as a black box. In the figure, we detail its internal view for illustration.

Because of the cycles in the data dependencies, there is no way to write the programs associated with the controller `sync1`:

- First, the wire `d` creates a cyclic dependency between an output of the component AB and an input of the same component.
- Second, there is another cyclic data dependency: Between the components AB and C, the two wires `e` and `f` describe this situation.

To be correct, all the cycles described above must be cut with a PRE or any Moore-style component. On the contrary, if the sub-components are expanded; it will be visible that the wire `d` does not create a cycle; and the cycle created by the wires `e` and `f` is indeed cut with a PRE component.

Lustre/SCADE compilers also face this problem of cutting all the cycles at each level of the hierarchy. To accept designs as the one described in Figure 4.8, their solution is to expand all the composed components and check whether cyclic dependencies exist or not.

Another solution is to apply the principle of *modular compilation* of Lustre programs [Ray88, PR09]. This approach tries to identify those components that may be grouped together without restricting possible cyclic dependencies. In the context of a component model like 42, we keep the *FAMAPASAP* principle; this forbids access to the internal details of a component. Instead of expanding components, or imposing particular choices for encapsulation, we equip components with explicit specifications as we shall see in Section 4.2.1.

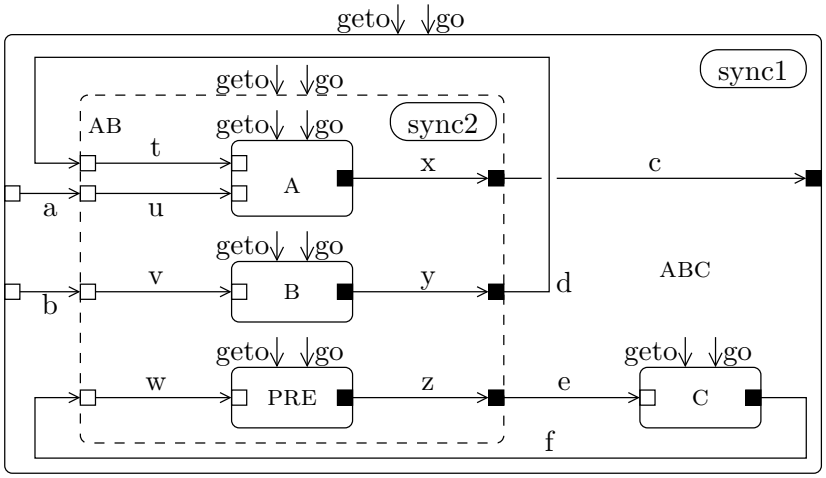


Fig. 4.8: Implicit specifications do not expose internal details

4.1.2 Simulation of Asynchronous Systems

In asynchronous models of computation, there is no explicit clock to synchronize the execution of components. Asynchronous systems range from threads on a mono-processor to large-scale multi-computer systems. Multi-processors systems are an intermediate case, where processors may run at distinct rates depending on the clock associated with each one. The clocks may be synchronized, but since they run in a true physical parallelism the most abstract view is to consider them as asynchronous.

Section 2.1.2 explains how to model asynchrony by interleaving components executions. Contrary to the synchronous *MoCC*, where one computation step of a component corresponds to one computation step of each of its subcomponents. A computation step in an asynchronous model corresponds to one computation step of some of its subcomponents.

Figure 4.9 is a simple model of a hardware architecture, very much in the spirit of so-called *transaction-level modeling* (see Section 2.2.1 for more details). The hardware platform being modeled is a simple multi-processor system. The two processors access the same memory via the bus. At any point in time, only one processor is allowed to access the memory. The connections between the hardware elements are not given in full detail as it would be the case at the register-transfer level (RTL). The exchanges between components are *transactions*[1], encapsulating the synchronizations that are necessary for one data exchange. A transaction is associated with two components: an *initiator* component that initiates the transaction and a *target* component that responds to it.

The example described in this section also serves as a simulation model. It is some abstraction of the behavior of the real system. The controller simulates the parallel execution of the processors, and their communication through the explicit memory. A simulation *MoCC* controller does not distinguish between components. This requires the components to expose the same control ports to such a controller. The controller activates components through their control inputs, and does not add any particular choice on their activation (e.g., synchronization, scheduling, etc.).

[1]The terms *transactions*, *initiators*, and *targets* are borrowed from TLM (see Section 2.2.1.1).

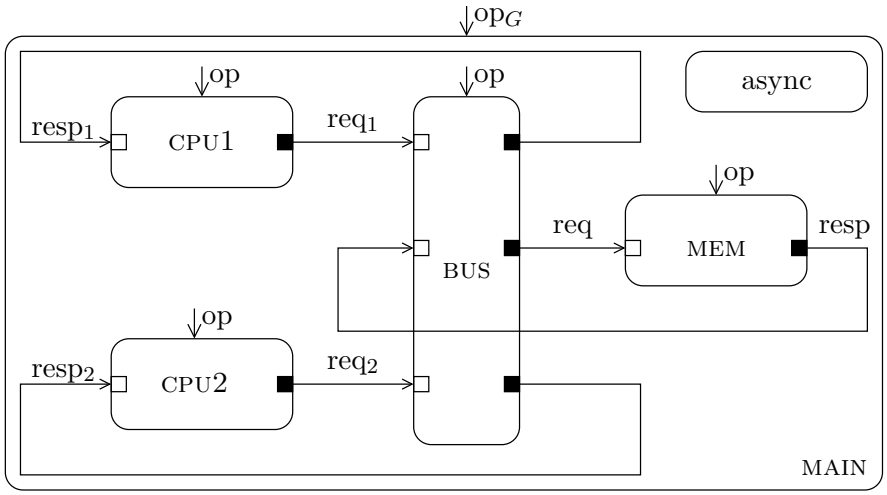


Fig. 4.9: A hardware simulation model

4.1.2.1 Individual Components and their Implicit Specification

The components communicate by means of transactions. The wires $\texttt{req}_x$ between the components are records encapsulating the information needed to initiate a transaction, i.e., the transaction type (read/write), the target address and the value to be written if applicable. Symmetrically, the response of a transaction is carried by the wires $\texttt{resp}_x$, which encapsulate the success of the transaction, and the value read in case of a read transaction.

Each component has a single control input `op`, to be activated by the simulation controller. The idea is that for each activation, a component representing a piece of hardware performs some internal computation, until yielding back control to the controller. For modeling such systems we require three classes of components. Each one with its own specification: Components initiating transactions, that is the two processors CPU1 and CPU2; components responding to transaction requests, that is MEM; and components carrying transaction requests and responses, that is the component BUS.

The component MEM models a memory which has one input data port `req`, and one output data port `resp`. It acts as a target component waiting for incoming transactions. When activated with the input control port `op`, it responds to an initiated transaction. It needs the parameters of the transaction on its data input `req` and delivers the data read and the result of this memory operation (SUCCESS or ERROR) on its data output `resp`.

The components CPU1, CPU2 model the behavior of distinct processors and have the same interface. The processors implement active behaviors, they are in charge of initiating transactions. The information needed for a transaction is sent through the output data port $\texttt{req}_x$. The processors receive a transaction response through the input data port $\texttt{resp}_x$. The input control port `op` is used to activate the component. When activated, a processor performs some computation until initiating a transaction through $\texttt{req}_x$. Once the response to the transaction is made available on its input port $\texttt{resp}_x$, the component should be reactivated with `op` to take it into account.

The component BUS manages the access to the memory, it its an abstraction of a communication bus as one can find in a typical *System-on-a-Chip*. It implements a simple behavior which consists in carrying the parameters of transaction from an initiator component to the target one, and the response to the transaction from the target to the initiator. It does not

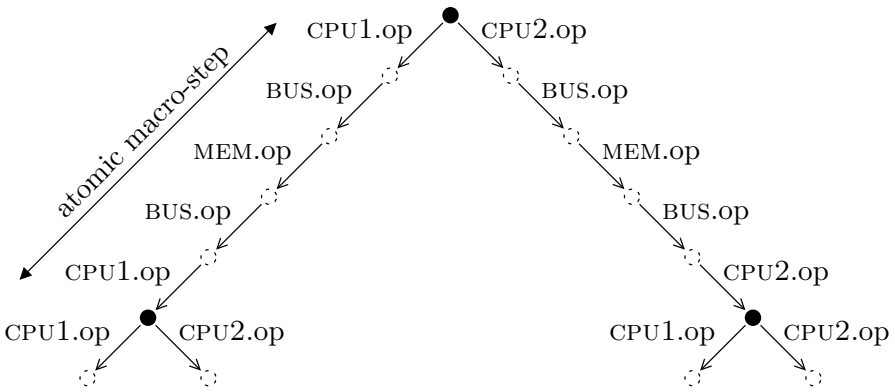


Fig. 4.10: The atomicity of a macro-step as exposed by the controller of Figure 4.11

implement any arbitration mechanism for concurrent accesses to the memory. The interface of the BUS exposes an input data port for each output data port of the other components, and a data output for each of their data inputs. For one transaction, it has to be activated twice with op; the first time to carry the parameters of the transaction to the target, the second one to carry the response from the target to the initiator.

4.1.2.2 The Controller

The 42 controller needed at this level is a *hardware transaction-level simulation* controller, i.e., a controller that simulates the potential physical parallelism between the hardware parts in a non-deterministic way. The controller exposes the behavior of components by interleaving their activations.

Figure 4.11 illustrates the code of the controller associated with the assembly of Figure 4.9. An activation of the component MAIN with $\mathtt{op}_G$ is translated by the controller into a sequence of micro-steps to perform a complete transaction: the controller randomly selects a processor to activate. The selected CPU performs some computation until requesting a transaction. The BUS is then activated to carry the parameters to MEM. When activated, MEM puts a response on its output data port which is carried by the bus to the corresponding processor. This requires another activation of the BUS. At the end, the selected CPU is activated in order to take the transaction response into account.

Figure 4.10 illustrates the atomicity of a macro-step in the simulation model. The black-filled circles denote the beginning and the end of a macro-step. The controller guarantees an initiated transaction to be atomic. That is, the activation of the component MAIN corresponds to one transaction, from its initiation by one of the processors, until its termination.

Successive activations of the component MAIN result in the interleaving of transactions, in the same spirit of the interleaving of asynchronous systems with shared memory described in Section 2.1.2.

4.1.2.3 Comments

The example is a model of an asynchronous system with shared memory. It models the asynchronous execution of the two processors and their synchronous communication with the explicit memory. That is, each macro-step consists of the activations of a CPU, the BUS and the MEM. Hence, what we observe for a sequence of activations of the component MAIN is an interleaving of transactions. Because of the explicit memory MEM, there is no need for the implicit memory associated with the data ports of the components (see Section 3.1.3).

```
Controller async is {
  type request  = record {rw : bool; add, val : int};
  type response = record {res : bool; val : int};

  for op_G do : {
    var req, req_1, req_2 : fifo(1, request);
    resp, resp_1, resp_2 : fifo(1,response);

    int cpu := random(1,2) ;

      // CPU1 is selected for
      // a transaction with MEM

       if (cpu = 1){
         CPU1.op;
         req_1.put; req_1.get;
         BUS.op; req.put; req.get;
         MEM.op; res.put; resp.get;
         resp_1.put; resp_1.get;
         CPU1.op;
       }

      // CPU2 is selected for
      // a transaction with MEM

      if (cpu = 2){
         CPU2.op;
         req_2.put; req_2.get;
         BUS.op; req.put; req.get;
         MEM.op; res.put; resp.get;
         resp_2.put; resp_2.get;
         CPU2.op;
       }}
```

Fig. 4.11: Code of the asynchronous controller of Figure 4.9

The controller describes coarse granularity of simulation steps. This prevents some states from being observed. For example, the state where the two processors request a transaction at the same time is not observable. To be able to observe such a detail, we should use a more detailed architecture of the hardware platform which requires a model of the bus that performs arbitration when needed.

Choosing the best granularity is an intrinsic modeling problem. If the granularity is too coarse, there are some behaviors that could be missed; if it is too fine-grained, the simulation is too slow. The great debate is not on how to tune the granularity of a simulation but how fine-grained it should be.

The example is an abstract model. It does not expose the details one can have in RTL models of the hardware. Notice that the details we gave in our model of the bus places it in the same category than the so-called TLM-programmer's view advocated for SoC design (see [CMMC08] for a discussion on these levels).

4.1.3 Hardware/Software Modeling

Figure 4.12 gives the details of the processor CPU1 of the previous example. The two processes running on CPU1 are modeled as sub-components of it. Processes evolve in parallel, independently form each other. Irrespective of the hardware platform, there is often a scheduling policy that allows processor and resources sharing between the processes. Depending on the scheduling policy, the scheduler elects the processes to execute. Non-eligible processes are processes waiting for an event to occur; e.g., a shared resource to be released, a value to be computed from another process, etc.

Process synchronization becomes a relevant issue for the design of such systems. There are many algorithms in the literature to implement synchronization mechanisms such as those used

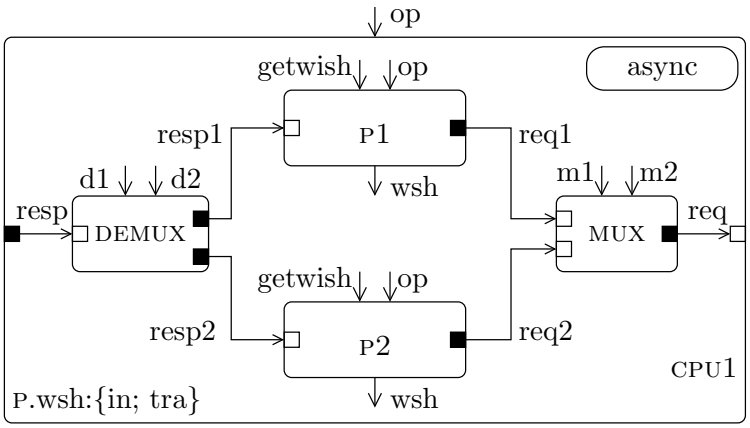


Fig. 4.12: Processes P1 and P2 running on the processor CPU1 of Figure 4.9

for mutual exclusion and rendez-vous. If synchronization is needed, processes should implement such algorithms.

4.1.3.1 Individual Components and their Implicit Specification

Each process may read (resp. write) something in memory by means of transactions. As described in the previous section, this needs a control information, an address, and the data to be written if applicable. The two processes initiate transactions through the same output data port of the processor. This implies a MUX component. Similarly, the input port of the processor is connected to the two input ports of the processes, which needs a DEMUX component.

A process component can be written in any imperative-style language, encapsulated so that it exports the data ports for the communication with the memory, two input control ports, and one output control port. The input control port `getwish` is used to ask it what it is willing to do. The process answers through `wsh` with a value `in` if the next execution step does not require access to the memory; or with the value `tra` if the process is willing to send a transaction. The input control `op` asks the process to execute one of its atomic operations. In the case of a transaction, it will output transaction parameters on its port `req`, the process should be re-activated when the response is available on its input `resp` in order to take into account the response of the transaction.

The MUX and DEMUX components are used to route data, they are controlled by control inputs `m1, m2, d1, d2` to choose the route.

4.1.3.2 The Controller

At this level, the 42 controller is not a hardware simulation controller. Still, it implements an asynchronous *MoCC*. This controller represents two things:

- an abstraction of an operating-system scheduler running on the processor.
- an abstraction of what happens in a real processor when it runs a piece of embedded software, taking its inputs from the memory, and writing its outputs to it. It shows how the processor writes to, and reads from the memory; it also transmits the data to and from the software.

The controller has two states labeled with `exec, wait` (see Figure 4.13). For each global activation it executes the code associated with one of these states. The controller implements the behavior of the encapsulating component (i.e., CPU1). We recall that the processor has to be activated twice. The first activation performs computation until sending a transaction (this

```
Controller async is {
  type request  = record {rw : bool; add, val : int};
  type response = record {res : bool; val : int};
  var  state : {exec, wait} := exec;
  pwsh : {in, tra};
  p: int // the running process
  for op do : {
   var req, req1, req2 : fifo(1, request);
   resp,  resp1, resp2 : fifo(1,response);

    switch(state){

          case exec:                          case wait:
           transaction: bool :=  false;        resp.put;
           while(! transaction){               resp.get;
             p := random(1,2) ;                if(p=1){
             if(p=1){                           DEMUX.d1;
               P1.getwish;                      resp1.put;
               pwsh := P1.wsh;                  resp1.get;
               if(pwsh = in) {P1.op;}           P1.op;
               else{ // pwsh = tra              }else //(p=2){
                 P1.op; req1.put; req1.get;     DEMUX.d2;
                 MUX.m1;                        resp2.put;
                 req.put; req.get;              resp2.get;
                 transaction :=true;            P2.op;
               }                                }
             }else //(p=2){...}                 state := exec; break;
             }                                  }
             state := wait; break;          }
```

Fig. 4.13: Code of the asynchronous controller of Figure 4.9

corresponds to the state `exec`). The second takes into account the transaction response (this corresponds to the state `wait`).

Figure 4.14 illustrates the micro-steps associated with two successive activations of the component CPU1. The black-filled circles denote the beginning and the end of the micro-steps. Each circle is associated with a label to indicate the state of the controller at that circle.

At state `exec`, the controller acts as a scheduler. It selects a process randomly; the global variable `p` identifies the selected process. To activate the selected process, the controller first asks it whether it is about to perform an internal move, or a read/write. Depending on the answer `wsh`, it asks the process to perform a single operation, or asks it to send the transaction parameters which are routed to the global output thanks to the component MUX. Notice that the activation of the processes is inside a loop. The break condition of the loop is the value of the Boolean variable `transaction`. The controller interleaves the activation of the processes inside the loop until one of them decides to send a transaction. This way the controller complies with the specification of the component CPU1 describes in Section 4.1.2.1. In Figure 4.14, this corresponds to the sequence of micro-steps between the state `exec` and `wait`.

At state `wait` the controller is waiting for a response of the pending transaction. When it is activated with `op`, a data is supposed to be available on its input `resp` (implied by the specification of the component CPU1). This data is routed thanks to the component DEMUX to the process that initiated the transaction. This process is identified with the global variable `p` inside the controller. In Figure 4.14, this corresponds to the sequence of micro-steps between the state `wait` and `exec`.

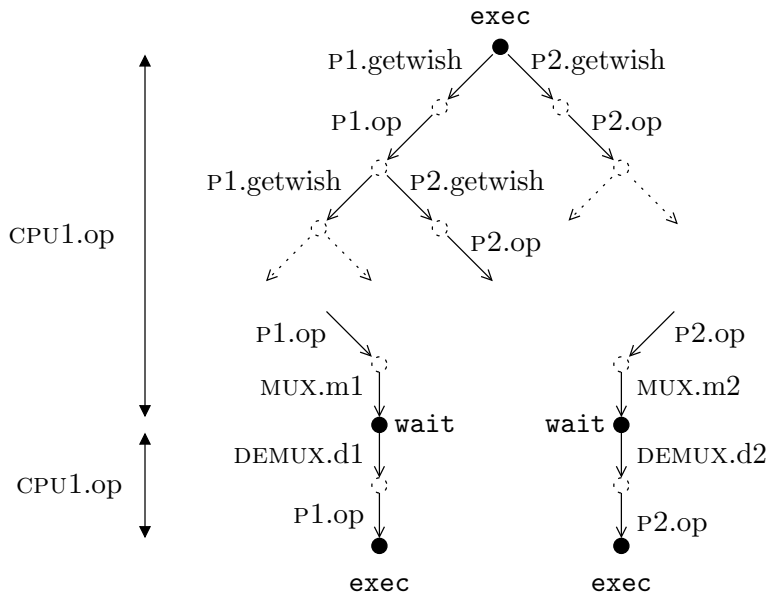


Fig. 4.14: The micro-steps associated with two successive activations of CPU1

The execution of the code associated with `exec` or `wait` depends on the variable `state`. When CPU1 is instantiated somewhere, this variable is initialized to the value `exec`. After each activation, the value of the variable `state` alternates between `exec` and `wait` so that the component CPU1 exposes the cyclic behavior: `sending transaction, waiting for response, sending transaction, etc.`.

4.1.3.3 Comments

The example models Hardware/Software components. It illustrates how we deal with heterogeneity in 42, with hierarchic levels, each level using a particular *MoCC*. It especially deals with hardware/software heterogeneity.

The controller plays the role of a scheduler. More precisely, it mimics the behavior of a non-preemptive scheduler. To model preemptive schedulers, the code of the process should yield after each piece of code which is guaranteed to be atomic (non interruptible) by the hardware, or by language features like the `synchronized` keyword in Java.

The scheduling policy is implemented by the controller. The scheduling policy is based on non-deterministic interleaving of components. When modeling systems with more complex scheduling policies, one may consider the scheduler as a component part of the functional specification of the system. The 42 controller should be here only for simulation. It interleaves components' execution taking into account the scheduler decisions to decide which process to activate.

Modeling asynchronous behaviors is interesting. It can be used to study the behavior of algorithms used in asynchronous parallel programming, like the Peterson algorithm for mutual-exclusion. The code of the controller can be used to produce all the possible interleavings of the two processes, together with their effect on the memory.

4.1.4 Kahn Process Networks

Figure 4.15 shows a simple example of a Kahn Network [Kah74, KB77]. In such a model of computation, the processes produce outputs depending on the inputs they have. The process `P1`

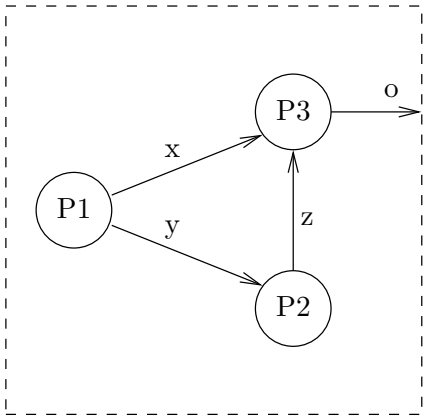


Fig. 4.15: An example Kahn Process Network

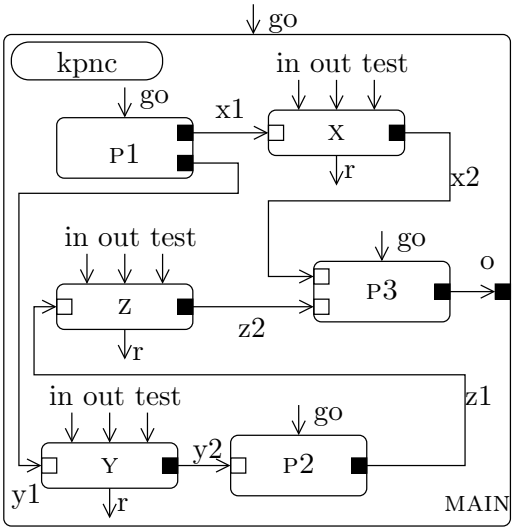


Fig. 4.16: The 42 view of the KPN example

is a producer, it produces continuously new values on its outputs. The processes are supposed to execute in pure asynchronous parallelism. They communicate through channels (`x,y,z`). Theoretically, the channels are associated with unbounded FIFOs. Writing on a channel is non-blocking while reading from it is blocking. This means that a process waiting for an input cannot execute. Moreover, a process cannot test its inputs channels for emptiness.

4.1.4.1 Individual Components and their Implicit Specification

The description of Kahn Networks [Kah74] in 42 is an interesting exercise which shows the main difference between the potential memory present as a component, and the volatile memory associated with the wires for a particular *MoCC*. The memory associated with the wires serves only as temporary storage to build a macro-step (see Section 3.1.3); it is initialized at the beginning of each macro-step.

To describe a Kahn Network in 42, we need two types of components: components modeling the processes, and components modeling communication channels implemented as explicit unbounded FIFO queues between the processes.

The processes (P1, P2, P3) have input and output data ports. Each of them is equipped with an input control port `go` which means: read the inputs and provide the outputs. The processes are associated with the implicit specification: each activation needs all the inputs to compute all the outputs.

The channel components have an explicit control input `test` that allows to test them for emptiness; the answer `r` is an explicit control output of Boolean type. Writing true on the control output `r` means that the channel contains at least one element. The idea is that this information may be used by the controller (which describes the semantics of KPN) but not by the components themselves, because components are not allowed to test the input channels for emptiness. The channel components also have two control inputs: `in` (to accept a value) and `out` (to deliver a value).

4.1.4.2 The Controller

The controller should be able to expose the potential parallelism of the processes. Following the definition of the KPN *MoCC*, it should also describe the fact that a component modeling a

```
Controller kpnc is {
for go do {
var x1,x2,y1,y2,z1,z2,i,o : fifo(1,int)
    ex, ey, ez : bool;

  int process := random(1,3);


  switch(process){
    case 1: P1.go;
            x1.put; x1.get; X.in;
            y1.put; y1.get; Y.in;
    case 2: Y.test;
            ey:=Y.ry;
            if(ey) then{
              Y.out;
              y2.put; y2.get; P2.go;
              z1.put; z1.get; Z.in;
            }
    case 3: X.test; Z.test;
            ex:= X.r;
            ez:= Z.r;
            if(ex && ez) then{
              X.out; x2.put; x2.get;
              Z.out; z2.put; z2.get;
              P3.go; o.put; o.get;
            }
          }
        }
```

Fig. 4.17: The programs of the controller

process is blocked until the values required on its inputs are available.

The code of the controller is illustrated in the Figure 4.17. The variables `ex, ey, ez` are used to store the answer of the channels after the test of emptiness (i.e., the activation with `test`). It associates a one-place integer FIFO with each wire connecting two components. These FIFOs are initialized at each global activation.

The controller simulates the true parallelism between processes by interleaving their activations. For each activation of the component MAIN, it randomly selects a process to be executed. Once a process is selected, the controller verifies that all its inputs are available, by testing the input channels. If it is ok, it takes the inputs in the corresponding channels; the process is activated with `go` to produce all its outputs, which are stored in the corresponding channels. If one of the channel components connected to the inputs of the process answers with `false` on its output `r`, the process is not activated with `go`. This describes the blocking read of the process.

4.1.4.3 Comments

The memory associated with the wires is not persistent. The example insists on the lifetime of the memory associated with the wires by describing the difference between it and the components acting as a real memory (i.e., the components modeling the channels).

4.1.5 Globally Asynchronous Locally Synchronous Systems

Figure 4.18 is an example *GALS* (Globally Asynchronous Locally Synchronous) modeled with 42. The system involves two different *MoCCs* (synchronous and asynchronous) organized in a hierarchy. The example involves two synchronous components SYN1 and SYN2 (see Section 4.1.1) which are run asynchronously, i.e., they have no common clock. This requires the use of buffer components to store the information produced before it is used. For instance, SYN1 reads (at each tick of its own clock) the last value stored in BUF2 and produces an output which will be stored in BUF1. Independently, SYN2 does the same with respect to its clock.

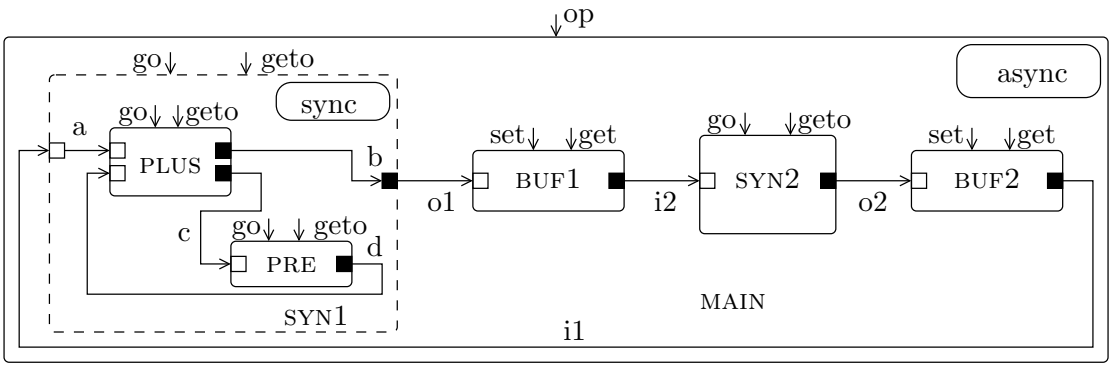


Fig. 4.18: Globally asynchronous locally synchronous system

4.1.5.1 Individual Components and their Implicit Specification

SYN1 and SYN2 are synchronous components. The details of the implementation of SYN1 are given for illustration. It is made of a PLUS and a PRE component to form an integrator (slightly distinct from the one described in Section 4.1.1 but exposing the same behavior). Their specification complies with what was described in Section 4.1.1. We recall that a clock cycle of a synchronous component consists of an activation with `geto` to compute the outputs, followed by an activation with `go` to change internal state. The activation with `geto` requires all inputs to compute all the outputs.

The components BUF1 and BUF2 are one-place buffers. Their memory is initialized to some value when they are instantiated. When activated with `set`, a buffer stores the value of its input port in its memory. When activated with `get`, the buffer delivers the last value stored in its memory on its output.

4.1.5.2 The Controllers

At the highest level of the hierarchy the controller `async` implements an asynchronous *MoCC*. Each global step `op` of the component MAIN is translated by the controller into a sequence of activations of the buffers and one of the synchronous components.

The code of the controller `async` is given in Figure 4.19. Asynchrony is obtained by random selection of the component to execute. The variable `x` is assigned a random value, it defines whether we consider an occurrence of SYN1's clock or that of SYN2. For `x=1` the controller activates `BUF.get` which will deliver the input for SYN1, activates the latter with `SYN1.geto` to produce the output and with `SYN1.go` to change its state. Finally the micro-step `BUF.set` will store the output in BUF2.

The controller `sync` implements a synchronous *MoCC*. It is associated with the assembly defining the architecture of the component SYN1. Its code is illustrated in the Figure 4.19. As described in Section 4.1.1, in response to the activation with `geto` it activates (with `geto`) the subcomponents of SYN1 in an order compatible with the data dependencies. The activation of components SYN1 with `go` is translated into a sequence of activations with `go`.

4.1.5.3 Comments

Mixing distinct *MoCCs* is the key of modeling heterogeneity. The example illustrates how to put in a hierarchy distinct *MoCCs* in order to model heterogeneity with 42.

The example considers unrelated clocks. There are no constraints on asynchronous scheduling of synchronous components (periodicity, priority, etc.). There may be loss

```
Controller async is{
  for op do {
    var i1, o1, i2, o2 : fifo(1,int);
    int x := rand(1,2);
    if(x=1)then{
      BUF2.get; i1.put; i1.get;
      SYN1.geto; o1.put; o1.get;
      i1.put; i1.get; SYN1.go;
      BUF1.set;
    }else{
      BUF1.get; i2.put; i2.get;
      SYN2.geto; o2.put; o2.get;
      i2.put; i2.get; SYN2.go;
      BUF2. set;
    }
  }
}
```

(a)

```
Controller sync is {
  for geto do {
  var a,b,c,d : fifo(1,int);
    PRE. geto; d.put; d.get;
    a.put; a.get; PLUS.geto;
    b.put; b.get;
  }

  for go do {
  var a,b,c,d : fifo(1,int);
    c.put; c.get;
    PRE. go;
    a.put; a.get;
    d.put; d.get;
    PLUS.go;
  }
}
```

(b)

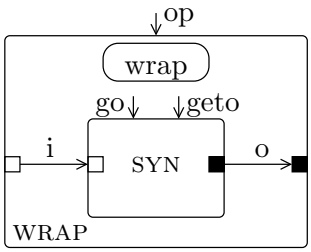
Fig. 4.19: Asynchronous (a) and Synchronous (b) controllers of the GALS system of Figure 4.18

of information (SYN1 may write a new value on BUF1 before SYN2 may read the old one). The example is intended to demonstrate heterogeneity modeling with 42 components, not to reason on detailed implementation.

The modeling of the asynchronous execution of several synchronous components (Figure 4.18) should be faithful to reality, that is, the clocks are *not synchronized*. This is why each activation of MAIN with `op` corresponds to one execution step (a clock cycle) of one of the two synchronous components only. In this kind of model, there is no relation between the clocks.

Wrapper controllers may be used to interface distinct *MoCCs*. When mixing controllers, there may be a need for wrapping components into an interface complying with the specification of components of other *MoCCs*. For instance, to make the synchronous components compliant with the asynchronous components described in Section 4.1.2, one would wrap them into a component having only one control input `op`.

The controller inside the wrapper is in charge of translating one activation with `op` into an activation of the synchronous component with `geto` followed by an activation with `go`. Figure 4.20 illustrates such a wrapping, and the controller `wrap` associated with the assembly. For instance, we can replace SYN1 and SYN2 by their wrapped version in the system of Figure 4.18 in order to simplify the code of the controller `async` of Figure 4.19. In this case, the controller interleaves component executions only. It does not manage their activation with `geto` and `go` in order to perform a clock cycle; this is left to the controller `wrap`.




```
Controller wrap is {
  var i,o : fifo(1,int);
  for op do {
    i.put; i.get; SYN. geto;
    i.put; i.get; SYN. go;
    o.put; o.get;
} }
```

Fig. 4.20: Wrapping a synchronous component

4.2 Examples with Explicit Contracts

4.2.1 Mono-Clock Synchronous Programs or Circuits

The example of Figure 4.21 illustrates an assembly of synchronous components. In this example, instead of considering components with implicit specifications, we associate a control contract with each component.

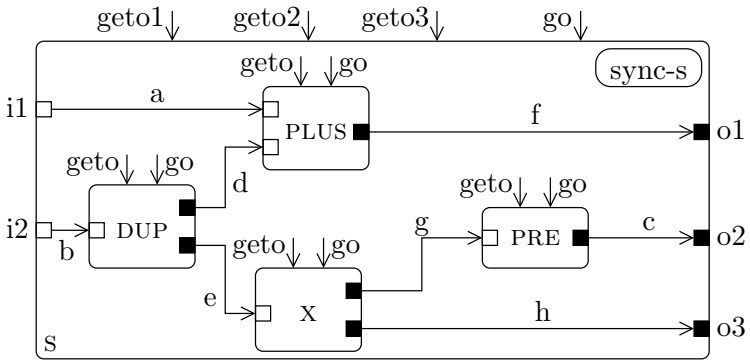


Fig. 4.21: An example of synchronous program

4.2.1.1 Individual Components and their Contracts

The system of Figure 4.21 is made of an assembly of synchronous components. The interface of each of them complies with what was described in Section 4.1.1. The encapsulating component S has more control inputs; this will be detailed later. To use these components, one may rely on the implicit specifications described so far. But for this example, we equip each component with its explicit specification, in order to expose the exact input/output data dependencies. Explicit specifications allows to overcome the limitations of the implicit specifications (see Section 4.1.1.8), in particular for detecting the absence of cycles.

In the sequel we describe the contract of some of the components. The language of contracts was described informally in Section 3.2.2. We recall the main aspects of this language along with the description of each contract.

The Component PLUS has two data inputs and one data output. Its contract is described in Figure 4.22; it applies to the combinational synchronous components (e.g., DUP) at the difference of the numbers of inputs/outputs. It says that the activation of PLUS with `geto` requires all the inputs (i.e., `a,d`) to compute all the outputs (i.e., `f`). The activation with `go` has no data dependencies.

The Component X is assumed to be a sequential synchronous component, i.e., it manages some internal memory. Its contract (Figure 4.22) complies with the implicit specification we gave so far for those synchronous components behaving as *Mealy machines*.

The activation of the component X with `geto` requires the input `e` to compute the outputs `g` and `h`. The activation with `go` will update its internal state; it requires the input `e` for that purpose.

The Component PRE is a *Moore-Style* machine. Its contract is described in Figure 4.23. The activation with `geto` requires no inputs and provides a value on its output data port `c`.

Contract for PLUS | Contract for X

Fig. 4.22: Contracts for Mealy-style components PLUS, X

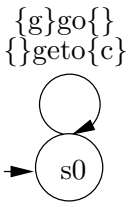


Fig. 4.23: Contract for the Moore-style component PRE

The activation of the component with `go` will change its internal state; it requires a value on its input data port `g`.

The Component S The components DUP, PLUS, X, and PRE are assembled to define the architecture of the component S. As we will show later, the decision to consider explicit specifications with the components allows for partial computation of the outputs. This is something that would be complicated (or even impossible) with implicit specifications because of the data dependencies between inputs and outputs.

Notice that the interface of the component S exposes two input data ports `i1, i2`; three output data ports `o1, o2, o3`; and four input control ports. For each output data port we associate a `geto` control input (`geto1, geto2, geto3`). To change state, we require only one control input `go`.

4.2.1.2 The Controller

Figure 4.24 shows the code of the controller `sync-s` associated with the assembly of components in Figure 4.21. It implements the control inputs of the component S. For the computation of each output data port, the controller associates a sequence of micro-steps with the corresponding input control port. It reads the global inputs required for the computation of the corresponding output data port, and activates the components involved in the computation of its value. The activation of the components is done in an order compatible with the data dependencies.

For instance, for the activation with `geto1`, the controller reads the inputs on `i1` and `i2` (i.e., `a.put; b.put;`), activates the components DUP and PLUS. Finally, the micro-step `f.get` defines the value of the global output data port `o1`. The computation of the global output `o2` requires no inputs because it depends on the component PRE; the component PRE requires no inputs to compute its outputs. The computation of the global output `o3` requires the global input `i2`.

The activation of the component S with `go` is translated by the controller into a sequence of activations of the sub-components with `go`. For the sequential components PRE and X, the controller takes care of providing them with their inputs before the activation. The values are taken from the output ports of X and DUP respectively, because the memory associated with

```
Controller sync-s is {

for geto1 do{
  var a, b...

  a.put; a.get;
  b.put; b.get;
  DUP.geto;
  d.put; d.get;
  PLUS.geto;
  f.put; f.get;
}

for geto2 do{
  var a, b...

  PRE.geto;
  c.put; c.get;
}

for geto3 do{
  var a, b...

  b.put; b.get;
  DUP.geto;
  e.put; e.get;
  X.geto;
  h.put; h.get;
}

for go do{
  var a, b...

  PLUS.go
  DUP.go
  g.put; g.get;
  PRE.go;
  e.put; e.get;
  X.go;
}}
```

Fig. 4.24: The code of the controller `sync-s` associated with the assembly of Figure 4.21

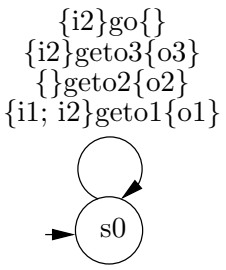


Fig. 4.25: An example contract for the component S of Figure 4.21

output data ports is persistent across the global activations (see Section 3.1.3).

4.2.1.3 A Control Contract for the Component S

The contract illustrated in the Figure 4.25 is a possible description of the behavior of the component S. With each activation $\texttt{geto}_i$ is associated a transition labeled with the data dependencies corresponding to it. For instance, `{i1; i2}geto1{o1}` describes the dependencies for the computation of the output `o1`. Notice that the component may be asked to compute its outputs in any order.

The component S is sequential; it encapsulates some memory associated with X and PRE. The `go` transition declares that the computation of the new state of S depends on `i2`, because the states of both X and PRE depend on `i2`.

The code of the controller `sync-s` and the control contract associated with the component S may be deduced from the architecture and the control contract of the components PLUS, X, etc. (see Section 6.2.1).

4.2.1.4 Benefit of Using Explicit Contracts

Figure 4.26 illustrates the architecture of the component XS. The component consists of an assembly of the component S and the component X (described above).

When we consider implicit specifications for each component, we would say that the computation of each output requires all the inputs. As commented in Section 4.1.1, one cannot write a controller for such an assembly because of the cyclic dependencies between the components.

In case we consider explicit specifications, we know that the cycle created by the wire `o3` is in fact not a cycle, because the computation of this output does not depend on the input to which

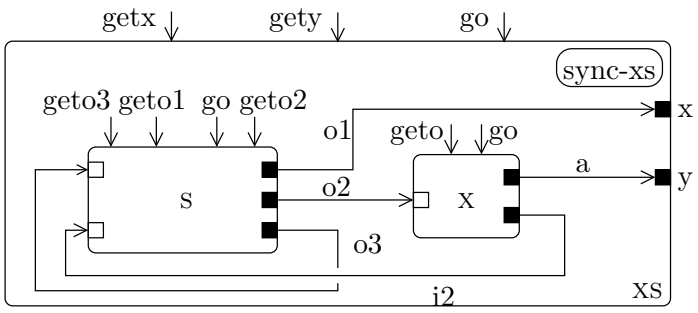


Fig. 4.26: Assembling components described with explicit specifications

```
Controller sync-xs is {

  for getx do{
    var o1,...
    S.geto2;
    o2.put; o2.get;
    X.geto;
    i2.put; i2.get;
    S.geto3;
    o3.put; o3.get;
    S.geto1;
    o1.put; o1.get;
  }

  for gety do{
    var o1,...
    S.geto2;
    o2.put; o2.get;
    X.geto;
    a.put; a.get;
  }

  for go do{
    var o1,...
    i2.put; i2.get;
    S.go;
    o2.put; o2.get;
    X.go;
  }}
```

Fig. 4.27: The code of the controller associated with the assembly of Figure 4.26

it is connected (i.e., `i1`). Moreover, we know that the cycle created by the wires `i2` and `o2` is in fact cut by a PRE component inside the component S. In the contract of S (Figure 4.25), the transition `{}geto2{o2}` expresses that the computation of the output `o2` depends only on the internal memory of the component S. This is synonym of the presence of a PRE component, which is the elementary memory point in synchronous systems.

One can write a controller (`sync-xs`) consistent with the contracts of the components S and X in order to define the behavior of XS. The code of such a controller is illustrated in Figure 4.27. This controller implements a synchronous *MoCC* and allows for partial computation of the outputs of the component XS. For instance, for the computation of the global output `x`, it activates the component S with the control input `geto2` without providing it with any inputs (this is specified by the contract of the component S). Then it activates the component X to provide the input `i2` to the component S. Now, S is activated with `geto3`; this provides the input to the component S such that it may be activated (with `geto1`) to compute the output `o1` connected to the global output data port `x`.

4.2.1.5 Comments

The contracts expose interesting details about components. Writing a control contract for a synchronous component enables the description of fine details related to its behavior. Essentially, the contract expresses partial input/output data dependencies to avoid looking at its implementation. Moreover, for a sequential component the contract also declares the required inputs to change internal state.

The computation of the outputs may be done in any order. The contract of S allows

for activating the component in any order. This is interesting because one may not wait for all the inputs to be available, and may compute an output when its required inputs are available. Moreover, it is possible to compute only part of the outputs and change state, for instance, in case we do not care about some outputs.

4.2.2 Multi-Cycle Synchronous Programs or Circuits

In a mono-clock synchronous program, all components execute together with respect to one common and global clock. At each clock cycle, each component reads its inputs, computes its outputs and updates its internal state. Sometimes, there is a need for systems where components execute at distinct rates, each of them performing computations only at some instants. These instants are defined by a clock associated with the component. Multi-cycle programs are systems where there are at least two distinct clocks. One of these clocks is the base clock. The other clocks are slower.

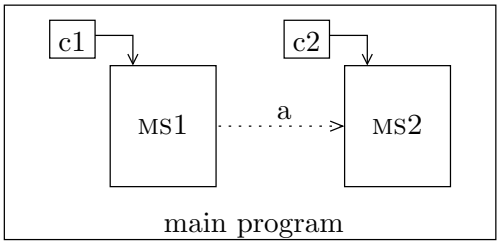


Fig. 4.28: Two communicating components running at distinct speeds

Figure 4.28 is a description of a multi-cycle system with an abstract notation (it is not a 42 model). The components MS1 and MS2 are two synchronous systems. Each one is associated with a clock (`c1` and `c2` respectively). Clocks are Boolean flows. For instance, the base clock is the Boolean flow taking the value `true` at each instant. At each cycle of the base clock, `c1` (resp., `c2`) takes a Boolean value that indicates if MS1 (resp., MS2) will indeed perform a computation step.

The component MS1 computes some data which is sent to the component MS2. As the components may not be synchronized, there may be a need for some explicit memory acting as a buffer to manage their communication.

In what follows, we present a type system for synchronous languages dedicate to clocks (Section 4.2.2.1). Section 4.2.2.2 is a discussion on the size of the buffer to use in order to manage components communication. In Section 4.2.2.3, we present the modeling of multi-cycle systems with 42, and how contracts may help in identifying the clocks associated with the components. The purpose of modeling such systems in 42 is to expose the potential memory used for the communication between components associated with distinct clocks.

4.2.2.1 Clock Calculus in Synchronous Languages

The compilation process of synchronous languages consists in more than simple code generation. It includes some analysis of the program such as data dependencies analysis and *clock calculus*. The latter is a *type system* for clocks. It consists in associating a clock with each expression of the program and checking the consistency of the clocks associated with the flows involved in an operation. The clocks associated with the operands of any operator must be synchronized, i.e., they take the same value at each instant.

Since static verification of the equality of two Boolean expressions is undecidable, clock checking in Lustre relies on syntactic substitutions to prove the equality of two clocks [CPHP87]. In

Signal, clock calculus consists in the synthesis of some constraints associated with clocks, and verifying their consistency [BLG90].

In 42, which is not a language for the design of synchronous programs, the clocks are not dealt with as in Lustre and Signal. In particular, there is no dedicated type system. The clocks in a 42 model are similar to the *activation condition* operator (*condact*) in SCADE.

4.2.2.2 Communication Between Components Requires Memory

The size of the buffer used to manage the communication between two components associated with distinct clocks depends on the behavior of the clocks. There may be four possibilities:

(1) **Synchronized clocks:** If the clocks are synchronized, i.e., they take the same value at each instant, the system may be considered as mono-clock synchronous system as in Section 4.2.1. One may avoid the use of buffers to make components communicate. The value produced by the producer is consumed at the same instant. This is the case for mono-clock synchronous programming in Lustre [HCRP91] for instance.

(2) **Clocks are not synchronized but somehow equivalent:** This case applies for systems where the difference between the amount of computed values and the amount of the values consumed is always positive and never exceeds the size of a buffer (`n`).

The *N-Synchronous* [CDE+06] model is an approach to the modeling of such systems. It is built over a synchronous formalism, and proposes a relaxed notion of synchrony. It aims at synchronizing flows associated with distinct clocks using intermediate buffers. The approach requires clocks to expose periodic behaviors [CDE+05], from which fixed-size buffers may be computed.

(3) **The clock of the producer is faster:** When the producer produces faster than the consumer can consume, putting a buffer to store values is no longer possible. Suppose for example that the producer is two times faster than the producer. The capacity of the buffer will be exceeded whatever the size of the buffer. For such kind of applications, we use one-place buffers which always deliver the last value stored, or their initial value. The effect of using such a buffer is to sample the values produced with respect to the clock of the consumer (e.g., the operator `when` in Lustre).

(4) **The clock of the consumer is faster:** Here, the right choice is to use a one-place buffer with an initial value. The buffer is supposed to hold the last value produced and deliver it whenever it is asked for. The buffer in this situation plays the role of a projection operator (e.g., the operator `current` in Lustre).

In what follows, we first illustrate the general modeling of multi-cycle synchronous systems in 42, then we discuss the size of the buffer being used with regards to the information we have about the clocks.

4.2.2.3 Modeling Multi-cycle Systems in 42

Figure 4.29 illustrates the modelling of the system of Figure 4.28 in 42. We can distinguish the components MS1 and MS2 (the internal details of MS1 are exposed, they are described later). The clock of each component is explicitly described as a data input of the component. `c1'` (resp., `c2'`) defines a clock for the component MS1 (resp., MS2). The clock values are of type Boolean, they are generated by the synchronous components CLK1 and CLK2. The components R1 and R2 are used by the controller to know about the values of the clocks (see Section 3.1.2.1); these components replicate their input on their output control and data ports. To manage the

communication between the components MS1 and MS2, we use an explicit buffer BUF. The size of the buffer varies according to the relation between the clocks of components.

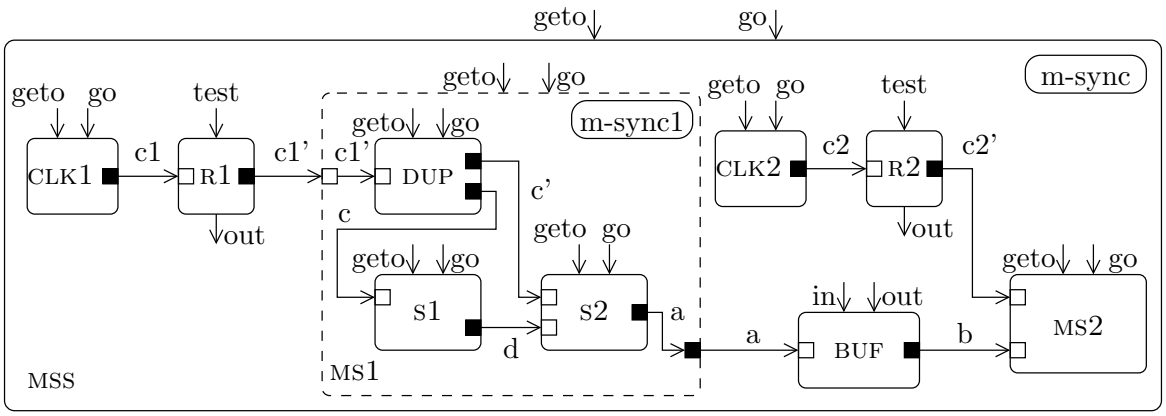


Fig. 4.29: One-place buffer used to manage multi-cycle components communication

The synchronous components expose the same interface as typical synchronous components described in Sections 4.1.1 and 4.2.1. However, we require that some input/output data ports be associated with the clocks. These particular input/output are of Boolean type. The contracts allow to distinguish them from the actual inputs/outputs used for data exchange (see below). In the sequel we use `T` and `F` to denote the values `true` and `false` respectively.

4.2.2.4 Individual Components and their Contracts

The Component MS1 exposes one input data port `c1'` which corresponds to its input clock and one data output `a` on which it may deliver some value. The contract of MS1 is described in Figure 4.30. The activation of the component with `geto` requires the input clock `c1'` with the value `T` (i.e., true) to provide a value on the output data port `a`. The activation with `go` to change state also requires the input clock with the value `T`.

Associating the explicit value `T` with the clock `c1'` distinguishes this input from the inputs/outputs used for data exchange (i.e., `a`). The constraint put on the value of `c1'` means that the component MS1 may react only if it is on its clock, i.e., if its clock has the value `T`.

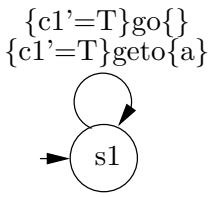


Fig. 4.30: The contract of MS1

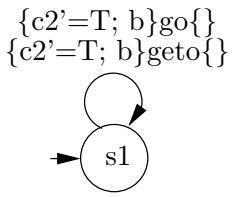


Fig. 4.31: The contract of MS2

The Component MS2 has two input data ports: `c2'` corresponds to its input clock, `b` is a data input. Its contract is illustrated in Figure 4.31. The `geto` (resp., `go`) activation requires the inputs `c2'` and `b`. Moreover, the value of `c2'` must be `T`, because the input `c2'` defines the clock of the component MS2.

The Components CLK1 ***and*** CLK2 are also synchronous components. Each of them has one Boolean output data port, `c1` and `c2` respectively. Their contracts are illustrated in Figure 4.32 and Figure 4.33 respectively. When the component CLK1 (resp., CLK2) is activated with `geto`,

it provides a Boolean value on its output `c1` (resp., `c2`). Its activation with `go` to change state requires no inputs.

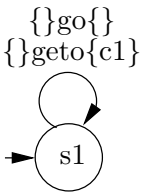


Fig. 4.32: The contract of CLK1

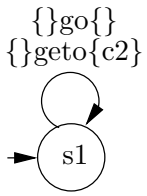


Fig. 4.33: The contract of CLK2

The Components R1 ***and*** R2 expose one control input `test`, one Boolean control output `vc`, one Boolean data input `c1` (resp., `c2`) and one Boolean data output `c1'` (resp., `c2'`). The contract of the component R1 is illustrated in Figure 4.35 (the contract of R2 is similar). When the component is activated with `test` it reads the input and reproduces its value on the output data and control ports.

The Component BUF acts as a buffer, its interface exposes one input data port `a` from wich it may take a value to store, and one output data port `b` on which it may deliver a value. The control inputs `in` and `out` are used to store and deliver a value respectively.

The contract of BUF is illustrated in Figure 4.34. It says that the component may be activated with `in` or `out`. The activation with `in` requires the input `a`; the activation with `out` requires no inputs and provides `b`. Notice that neither the interface of BUF nor its contract states on the size of BUF (i.e., one-place or n-places buffer). The size of the buffer is discussed in Section 4.2.2.6 below.

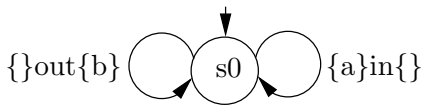


Fig. 4.34: The contract of BUF

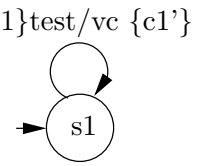


Fig. 4.35: The contract of R1

4.2.2.5 The Controller

The controller `m-sync` implements a *multi-cycle synchronous MoCC*. It defines the programs needed for the control inputs of the component MSS. It should be able to detect when a component may produce some outputs or require some inputs. To this end, it interrogates the components R1 and R2 through their control input `test`, and stores the value of their control output `vc` in α_1 and α_2, in order to know about the clocks associated with MS1 and MS2.

The controller `m-sync` is illustrated in Figure 4.36. It translates a global activation with `geto` into a sequence of activations of the synchronous components that are on their clock, with `geto`, in an order compatible with the partial order of data dependencies. The components CLK1 and CLK2 are always activated because they are implicitly associated with the base clock. The controller interrogates R1 (resp., R2) to know whether the component MS1 (resp., MS2) has to be activated. If MS1 is on its clock, the controller activates the component BUF with `in` in order to store the computed value. If MS2 is on its clock, the controller activates the component BUF with `out` in order to provide the component MS2 with the input.

The activation of MSS with `go` is translated by the controller into the activation of the synchronous components with `go`. The activated components must be on their clock.

```
Controller m-sync is {
  var  α1, α2 : bool

 for geto do: {
   var a, b : fifo(1,int);
   c1,...,c2' : fifo(1,bool);

   CLK1.geto; c1.put; c1.get;
   R1.test; α1 := R1.vc ;
   if(α1) then {
     c1'.put; c1'.get;
     MS1.geto;
     a.put; a.get;
     BUF.in;
   }
   CLK2.geto; c2.put; c2.get;
   R2.test; α2 := R2.vc ;
   if(α2) then {
     BUF.out;
     b.put; b.get;
     c2'.put; c2'.get;
     MS2.geto;
   }
 }
```

```
for go do{
  var a, b : fifo(1,int);
  c1,...,c2' : fifo(1,bool);

  CLK1.go;
  if(α1) then {
     c1'.put;
     c1'.get;
    MS1.go;
  }

  CLK2.go;
  if(α2) then {
     c2'.put;
     c2'.get;
     b.put;
     b.get;
     MS2.go;
  }
}
}
```

Fig. 4.36: The programs of the controller `m-sync`

4.2.2.6 Choosing the Size of the Buffer

Using a One-place Buffer In the example described above, the contracts of the components CLK1 and CLK2 expose no information about the values that the clocks `c1` and `c2` may take. The most general case is to consider a one-place buffer with initial value to manage the communication between the components MS1 and MS2. If the clock of MS1 is faster, then BUF acts as a sampling operator. At the opposite, if the clock of MS2 is faster, the buffer acts as a projection operator.

Using a N-places Buffer Figure 4.37 illustrates possible contracts for the components CLK1 and CLK2 where the values of the clocks are explicitly described in the contract. The behavior of the clocks is periodic, in the sense of [CDE+05]. Each four base-clock tick, `c1` takes the value `true` the two first tick, whereas `c2` takes `true` the two last ones. As we explained in Section 4.2.2.2-(2), as the clocks are somehow equivalent, we may use a n-places buffer. Precisely, a 2-places buffer, because the difference between the amount of consumed values and the computed ones never exceeds 2. The size of the buffer is computable from the description of the clocks, one may refer to the N-synchronous approach [CDE+06] for more details.

4.2.2.7 The Internal Details of MS1

The internal details of the component MS1 are described in Figure 4.39. It is composed of two multi-clock synchronous components S1 and S2, each one exposing one input data port associated with the clock of the component (i.e., `c` and `c'` respectively). The component DUP is a duplicator, it is used to duplicate the input clock of the global component MS2 (i.e., `c1'`) in order to provide the input clocks of the sub-components S1 and S2. This means that S1 and S2 are perfectly synchronized (the situation described in Section 4.2.2.2-(1)). In this case, there is no need for a communication buffer between S1 and S2; the components communicate through

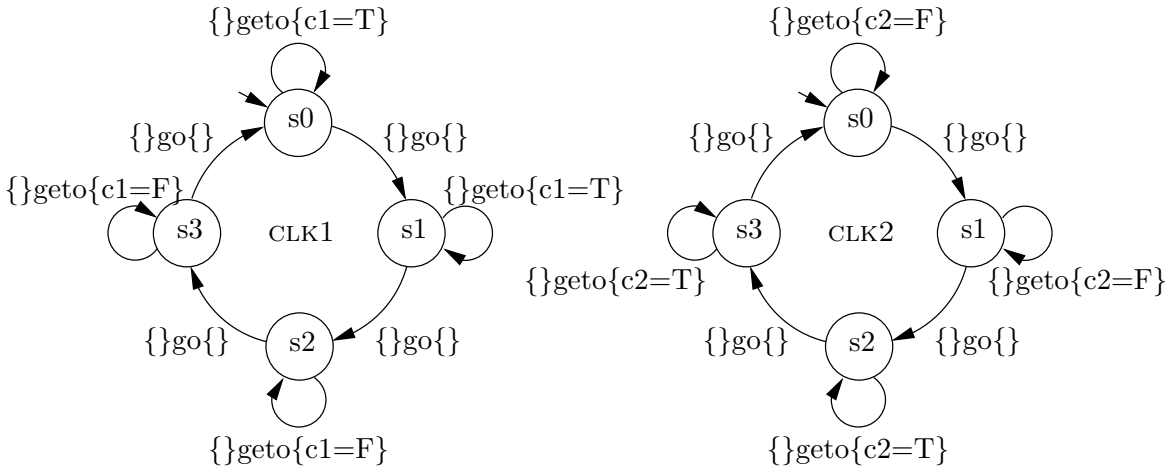


Fig. 4.37: The contracts of CLK1 and CLK2 describe the periodic behavior of c1: (TTFF)* and c2: (FFTT)*

the wire `d`.

The contract of each of these components is described in Figure 4.38. The component DUP is a combinational component, it requires all the inputs to provide the outputs. The contract of S1 (resp., S2) describes which of the inputs are the input clocks.

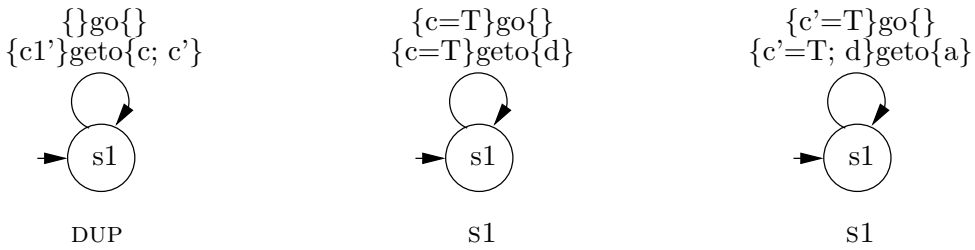


Fig. 4.38: The contract associated with the components DUP, S1 and S2

The Controller The controller associated with the component MS1 is illustrated in Figure 4.39. When MS1 is activated with `geto`, the controller activates the components DUP, S1, and S2 with `geto` in an order compatible with the partial order of data dependencies. The activation with `go` is translated into a sequence of activations of the subcomponents with `go`.

Notice that even if the components S1 and S2 are associated with clocks. The controller does not check their input clocks in order to decide on their activation. As the clock is a global input of MS1, it is the responsibility of the controller activating MS1 (i.e., `m-sync`) to check clock values. If `m-sync` uses MS1 according to its contract, MS1 is activated only if the clock `c1'` is `true`, hence, S1 and S2 are activated only if their respective clocks are `true`.

4.2.2.8 Comments

Modeling multi-cycle systems in 42. 42 is not a language for the design of synchronous systems. It allows for modeling various examples of multi-cycle synchronous systems, but there is no type system dedicated to clocks. Our modeling approach of clocks resembles to the activation conditions used in SCADE programs. Using 42 to describe multi-cycle systems has the benefit of modeling communication buffers explicitly as components.

Contracts and periodic clocks. When the clocks associated with the components are peri-

```
Controller m-sync1 is {
for geto do: {
  var c1', c, c': fifo(1,bool);
  a, d : fifo(1,int);

  c1'.put; c1'.get; DUP.geto;
  c.put; c.get; S1.geto;
  c'.put; c'.get;
  d.put; d.get; S2.geto;
  a.put; a.get;
}
for go do{
  var c1', c, c': fifo(1,bool);
  a, d : fifo(1,int);

  c.put; c.get; S1.go;
  c'.put; c'.get;
  d.put; d.get; S2.go;
  DUP.go
}}
```

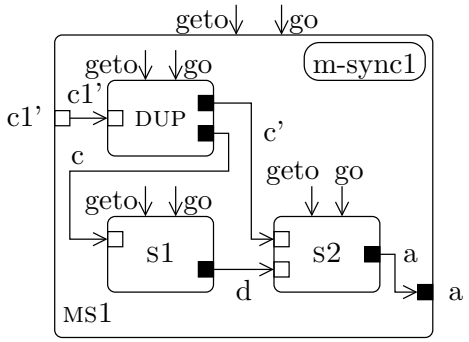
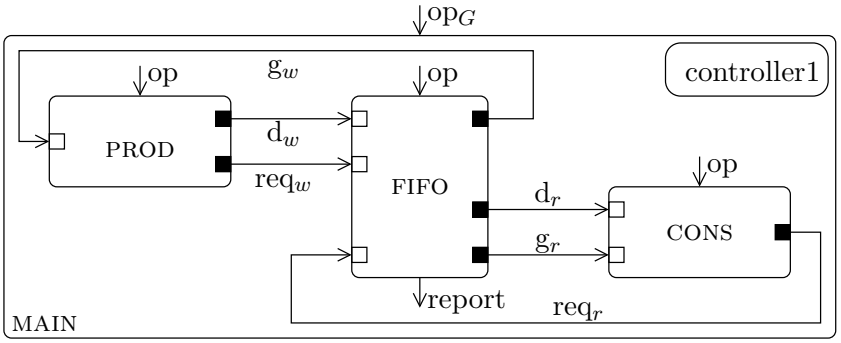


Fig. 4.39: The component MS1 and the programs of its controller `m-sync1`

odic, the contracts may be used to describe their behavior. In this situation, the contracts may be used in order to compute the size of the buffers. To this end, we can benefit from the work advocated to the N-synchronous approach.

4.2.3 Using Contracts to Describe Asynchronous Systems

The example of this section is intended to demonstrate how expressive the contracts are in describing components. In particular for describing their synchronizations in an asynchronous *MoCC*. Moreover, the example recalls the importance of the granularity in an asynchronous model, and shows how we can play with the code of a controller in order to observe more (resp., less) details during the simulation.



Ports Types :

$d_r, d_w : int$	$g_w, r_r : \{t, f\}$ (true, false)
FIFO.$report : \{ok, ko\}$	$req_r, req_w : \{t\}$

Fig. 4.40: The producer/consumer example in 42

Figure 4.40 shows the structure of a system made of three components: a producer, a consumer, and a bounded FIFO used to store the elements produced before they are consumed. The

intended behavior is that the producer and the consumer perform cyclic jobs, writing to or reading from the FIFO from time to time. The producer should wait when the FIFO is full, and the consumer should wait when it is empty.

In the 42 model, the data ports and connections are representative of the real hardware system. For instance, there is a protocol between the FIFO and the consumer: the latter should send a request to the FIFO to know whether it may deliver an element, and it is blocked until the FIFO answers this request by a grant signal. Similarly, the producer should send a request to know whether the FIFO still has some room available, and it is blocked until the FIFO accepts the WRITE operation by sending a grant signal.

Each component has a single control input called `op`, meaning: *perform a single atomic execution step*. The model is sufficiently abstract to represent systems in which the consumer and the producer are dedicated hardware components, or two CPUs with embedded software.

4.2.3.1 Tuning the Granularity of the Simulation

The controller we use with this example is an asynchronous simulation controller. It may be tuned in order to change the granularity of the simulation steps. We recall the importance of such a granularity in Section 2.1.2.

There are several cases for which this example may be of interesting uses. For instance, the model may be used in order to observe the communication between the consumer and the producer through the FIFO, or may be used in order to validate the communication protocol established between the FIFO and the other components. For the first situation, one may rely on coarse granularity of simulation steps. A simulation step in this case would be the complete dialog between the producer and the FIFO to write some data, or the complete dialog between the consumer and the FIFO to read some data. For the second situation, there is a need to observe more derails. For instance, we need to observe what happens when the consumer requests a read while the producer is writing to the FIFO. Hence the granularity must be finer.

In the sequel, we describe the components and their contracts in Section 4.2.3.2; we then describe two examples of controllers: one with coarse-grained simulations steps (Section 4.2.3.3), and another with fine-grained simulation steps (Section 4.2.3.4). Finally, we give some comments in Section 4.2.3.5.

4.2.3.2 Individual Components and their Contracts

The Producer's interface exposes two output data ports and one input data port: $\texttt{req}_w$ is used to send a write request; $\texttt{d}_w$ to send the data to be written; from $\texttt{g}_w$ it receives the grant to a write signal.

The contract of PROD is the one described in Figure 4.41. First it sends a request to write to the FIFO, via its port $\texttt{req}_w$. If it receives $\texttt{g}_w\texttt{=f}$, it is not allowed to write, and returns to its initial state (it will have to issue another request later). Otherwise, it will receive $\texttt{g}_w\texttt{=t}$. At this point, it may send a data via its port $\texttt{d}_w$ and return to the initial state.

The Consumer's interface exposes two input data ports and one output data port: $\texttt{req}_r$ is used to send a read request; from $\texttt{g}_r$ it receives the grant to a read signal; from $\texttt{d}_r$ it receives the read data.

The contract of CONS is the one in Figure 4.42. First it sends a request to read from the FIFO, via its port $\texttt{req}_r$, and waits for a response. If it receives $\texttt{g}_r\texttt{=f}$, it is not allowed to read, and returns to its initial state (it will have to issue another request). If it receives $\texttt{g}_r\texttt{=t}$, it is granted access to read; it also needs the data read, via its port $\texttt{d}_r$, and returns to its initial state.

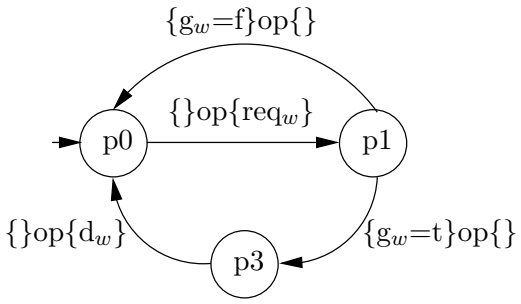


Fig. 4.41: Contract of the producer

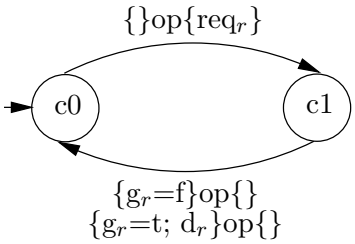


Fig. 4.42: Contract of the consumer

The FIFO's interface exposes one input port for each of the output ports of PROD and CONS; and one output port for each of their input data ports. Its contract is the one in Figure 4.43. First it receives a request for a read via its port `req`$_r$ and puts a value on the `report` output control port which is stored in variable α. This value tells the controller whether the FIFO is empty. If it is not empty ($\alpha = ok$), it grants access, providing `g`$_r = t$ together with the data read (via port `d`$_r$). Otherwise, if it is empty, it does not grant access and provides `g`$_r$`=f`. Similarly, the FIFO responds to a write request when it receives `req`$_w$. It provides a value on `report` stored in α telling whether the FIFO is full. If $\alpha = ko$ (the FIFO is full), it will not grant access for a write, and sends `g`$_w$`=f`. If $\alpha = ok$, the FIFO grants access for the write request (it sends `g`$_w$`=t`) and waits for the value to be written on its `d`$_w$ input data port.

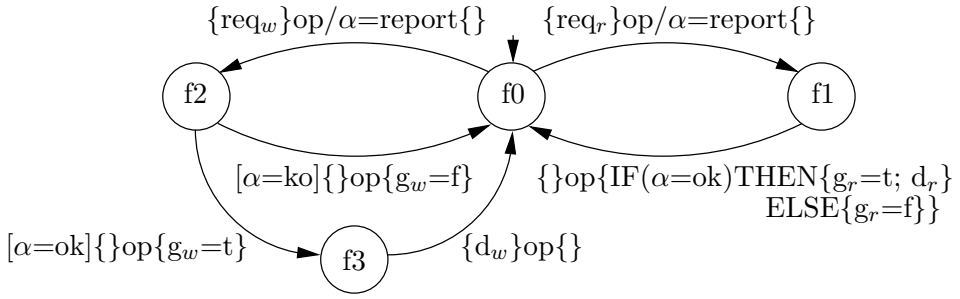


Fig. 4.43: The contract of the FIFO

4.2.3.3 A Controller with Coarse-grained Simulation Steps

Figure 4.44 illustrates the code of the simulation controller describing coarse-grained simulation steps. Basically, this controller does the following: for each global activation, it selects randomly one component from the producer and the consumer. If the producer is selected, the controller engages a sequence of activations of PROD and FIFO. It manages the dialog that happens between them, during a write request. The dialog will end by a success, i.e., PROD can write on the FIFO, or a failure, i.e., PROD can't write on the FIFO. In either case, the macro-step ends.

```
Controller controller1 is
var α:{ok, ko} ;
for op_G do : {
  var req_w, g_w, req_r, g_r: fifo(1, bool);
  d_w, d_r: fifo(1, int);

  int i := random(0,1);
  if(i=0){ // PROD writes to FIFO
    PROD.op; req_w.put; req_w.get;
    FIFO.op; α:= FIFO.report;
    if(α=ok){ // PROD can write
      FIFO.op; g_w.put; g_w.get;
      PROD.op; PROD.op;
      d_w.put; d_w.get; FIFO.op
    }
    if(α=ko){ // PROD can't write
      FIFO.op; g_w.put; g_w.get;
      PROD.op;
  }}
  else{ // CONS reads from FIFO
    CONS.op; req_r.put; req_r.get;
    FIFO.op; α:= FIFO.report;
    if(α=ok){ // CONS can read
      FIFO.op; g_r.put; g_r.get;
      d_r.put; d_r.get; CONS.op;
    }
    if(α=ko){ // CONS can't read
      FIFO.op; g_r.put; g_r.get;
      CONS.op;
   }} //end of macro-step
}
```

Fig. 4.44: Asynchronous simulation controller with coarse-grained macro-steps

Similarly, if the consumer is selected (the right part of the code), the controller manages the dialog that happens between CONS and FIFO components during a read request. The success or failure of the read request, ends the macro-step.

Figure 4.45-(a) illustrates part of the interleaving graph parsed by the controller. The black-filled circles denote the end and the beginning of a macro-step. Each macro-step consists of a set of component activations. For a sequence of activations of the component MAIN, we observe an interleaving of `READ` and `WRITE` actions from and to the FIFO.

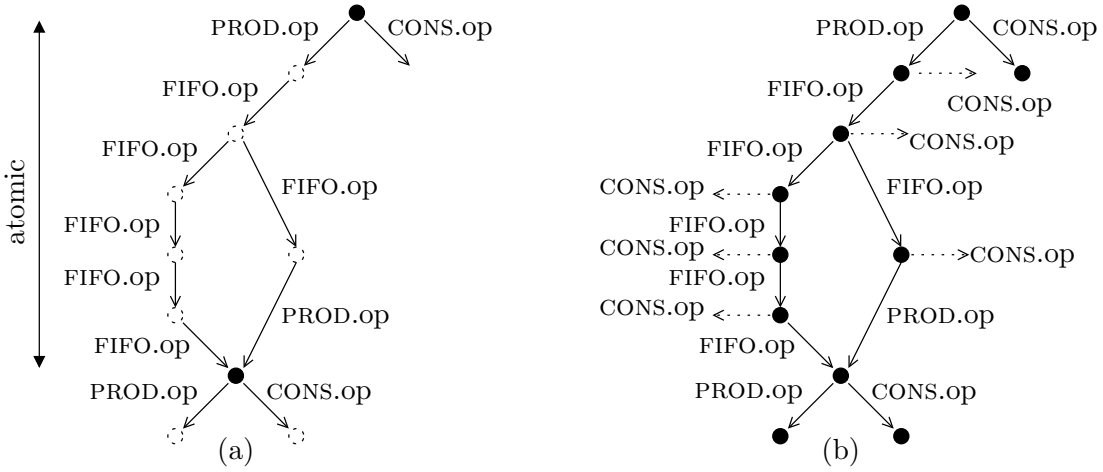


Fig. 4.45: Coarse-grained (a) Vs Fine-grained (b) macro-steps

4.2.3.4 A Controller with Fine-grained Simulation Steps

Figure 4.45-(b) illustrates the effect of considering fine-grained macro-steps comparing to the coarse-grained macro-steps described by Figure 4.45-(a). Each macro-step consists of exactly one activation of one subcomponent. As explained in Section 2.1.2.1, the refinement of the granularity in asynchronous models exposes more behaviors. In our example, this appears as

additional dashed arrows.

Figure 4.46 illustrates part of a controller with fine-grained macro-steps as described in Figure 4.45-(b). The granularity is defined by the transitions of the contracts because each macro-step consists of one subcomponent activation, which corresponds to one transition in its contract.

In order to be consistent with the contracts of the components, this controller must remember two things: the state of the contract of each component, and the produced outputs that are still not used. For instance, the variable $\texttt{p1\_f0\_c0\_req}_w$ in Figure 4.46 encodes the situation where the producer is at state `p1`, the FIFO is at state `f0`, the consumer is at state `c0`, and the output $\texttt{req}_w$ of the producer has been assigned a new value. At the beginning, the controller is at state `p0_f0_c0`. Each global activation, it changes to another state depending on the activated component, the inputs that were used, and the output that were produced.

```
Controller controller1 is {
var α : {ok, ko} ;
p0_f0_c0 : int := 0; p1_f0_c0_req_r : int: = 1; p1_f1_c0 : int := 2; ...
state : int := p0_f0_c0; c: int;

for op_G do : {
 var req_w, g_w, req_r, g_r: fifo(1, bool);
 d_w, d_r: fifo(1, int);

  switch(state){
    case p0_f0_c0:
      c := random(0,1);
      switch(c){
        case 0: PROD.op;
                state := p1_f0_c0_req_w;
                break;
        case 1: CONS.op;
                state := p0_f0_c1_req_r;
                break;
      }
      break;
                                        case p1_f0_c0_req_w:
                                          c := random(0,1);
                                           switch(c){
                                            case 0: req_w.put; req_w.get;
                                                    FIFO.op; ...
                                            case 1: CONS.op;
                                                    state := p1_f0_c1_req_r;
                                                    break;
                                          }
                                          break;
                                           case p1_f0_c0_req_r:
                                          ....
                                      }}}
```

Fig. 4.46: Asynchronous simulation controller with fine-grained macro-steps

4.2.3.5 Comments

42 control contracts are expressive. The contracts describe the communication protocol between the FIFO and the PROD or CONS components. Moreover, for some of the input/output data ports we expose explicitly their value (e.g., $\texttt{req}_w$, $\texttt{g}_w$, etc.). In fact, these data ports are part of the real system and are used for synchronization. Depending on the value carried by one of these data ports, a component may behave differently. Hence, their value should appear in the control contract of the components.

Playing with the granularity of the simulation. Contrary to various simulation models, tuning the granularity in 42 imposes some changes in the controller code not in the components themselves. This is a benefit of separating the simulation mechanics from the model semantics. In SystemC for instance, we need to change the implementation of components in order to tune the granularity.

The communication of asynchronous components requires memory. In the simulation model with coarse-grained steps (i.e., the controller in Section 4.2.3.3), only PROD and

CONS are asynchronous. They communicate through an explicit shared memory (i.e., the FIFO). In this case, there is no need for the memory associated with the outputs of the components.

At the opposite, in the fine-grained simulation (Section 4.2.3.4), all of PROD, CONS, and the FIFO are asynchronous. The required memory for the communication between PROD (resp., CONS) and FIFO is abstracted by the memory associated with the outputs of the components. The components PROD and CONS communicate through the explicit FIFO.

Automatic generation of asynchronous simulation controllers. The task of writing a simulation controller with fine-grained simulation steps would be complicated for complex simulation models. There may be several states to encode. In Section 6.2.2 we introduce controllers acting as *contract interpreters* to avoid the burden of writing them. The information required by these controllers is encoded in the contracts. Contract interpreters expose the same behavior as the controller in Figure 4.46.

Chapter 5

Formal Definition of 42

Introduction (En) *This chapter formally defines the various notions we presented so far. At first, we define the notion of components and how are they combined in order to form new components. Then, we define the operational semantics of the controller, and show how the behavior of a composed component is deduced from the behavior of its components and the programs of the controller. At the end, we formalize the contracts and the consistency relation between the components/controllers and the contracts.*

Contents

Introduction (Fr) *Ce chapitre décrit formellement les différentes notions que nous avons présenté jusqu'ici. Dans un premier temps, nous allons définir la notion de composant, et comment les assembler pour former des composants composites. Puis, nous définissons, à l'aide d'une sémantique opérationnelle, les différentes actions d'un contrôleur. Ensuite, nous montrerons comment le comportement d'un composant composite est déduit du comportement de ses sous-composants et les programmes associés au contrôleur. Enfin, nous formaliserons la notion de contrat est les différentes notions de compatibilité entre composants/contrôleurs et contrats.*

5.1 Components and Composing Components

This section deals with the formal description of the basic elements of the model introduced in Section 3.1. That is, the components, the architecture for combining them and the semantics of the controllers. We show at the end, how the behavior of a composed component may be deduced from the behavior of its sub-components, the wires describing its architecture and the controller for managing the execution of components.

5.1.1 Components and the Architecture Description Language

5.1.1.1 Components

A component has an *interface*: the set of input and output data ports, the set of input and output control ports. The ports take their values in some domain that can contain Boolean values, numerical values, etc. A component has an *internal state*, belonging to a set Σ. The most general definition of the behavior of a component is a set of relations corresponding to its possible activations through its control inputs. For each control input, the component behavior (which may be non-deterministic) is given as a relation that relates values for some of the data inputs, the current state, values for some of the control and data outputs, and a new state. Let us note $\mathcal{D}$ the union of all data types. We note $\perp \in \mathcal{D}$ an element that will be used whenever we refer to an undefined value. The *partial* valuations of the interface ports are represented by partial functions to $\mathcal{D}$. We note $f \in X \rightsquigarrow \mathcal{D}$ a partial function f, and $dom(f) \subseteq X$ its domain.

Definition 1 (Components) *A 42 component is a tuple:* $C = (\Sigma, \Sigma^{init}, IC, OC, ID, OD, \mathcal{B})$ *where* Σ *is the set of internal states,* $\Sigma^{init} \subseteq \Sigma$ *is the set of initial states, (one initial value is chosen when the component is instantiated) and* IC*,* OC*,* ID*,* OD *are the sets of names for the control inputs, control outputs, data inputs, data outputs, respectively.* $\mathcal{B}$ *is the behavior of the component, it is a total function* $\mathcal{B} : IC \longrightarrow \mathcal{R}$ *where* $R \in \mathcal{R}$ *is a relation:* $R \subseteq (\Sigma \times (ID \rightsquigarrow \mathcal{D}) \times (OD \rightsquigarrow \mathcal{D}) \times (OC \rightsquigarrow \mathcal{D}) \times \Sigma)$.

5.1.1.2 The Architecture

Components are assembled to form a *system* which may have global input and output, control and data ports. The architecture description language used to describe a system has a data flow style. A wire may relate an output data port of a component to the input data port of another (potentially the same), a global input data port to the input data port of a component, and an output data port of a component to a global output data port of the system. Wires do not mean a priori any synchronization, nor memorization, they describe how data flows from one data port to another.

Definition 2 (Architectures) *An architecture for combining a set of components* $\{C_i = (\Sigma_i, \Sigma_i^{init}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$ *is a tuple* $A = (IC_g, OC_g, ID_g, OD_g, L)$ *where the first two fields describe the control ports of the assembly, the two successive fields describe the data ports of the assembly, and* L *is the set of directed links between the data ports of the components, or between the data ports of the assembly and the internal ones:* $L \subseteq (\bigcup_I OD_i) \times (\bigcup_I ID_i) \cup ID_g \times (\bigcup_I ID_i) \cup (\bigcup_I OD_i) \times OD_g$*. Note that* $(x, y) \in L \wedge (x, z) \in L \Longrightarrow y = z$ *because links are point-to-point. Similarly* $(y, x) \in L \wedge (z, x) \in L \Longrightarrow y = z$*. The input and output control ports are implicitly linked to the controller.*

5.1.2 Controllers

A *system* made of components connected by wires has no semantics. The behavior of the system is defined by a *controller* to which are connected (implicitly) the control ports of the components. The controller activates the components, reads their control outputs and decides what happens on the wires.

Let us consider a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$, and an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$, defining a new component C_g.

The controller has some internal memory in the set Σ_C that can be used across the various activations of C_g (on the examples of Chapter 4, this corresponds to a control point in the program, and to the controller variables which are not attached to the links). It also has some internal memory associated with the wires $\Sigma_L : L \longrightarrow M$ that is reinitialized for each new activation. M is the union of the FIFO types.

The controller associates with each global control input $\text{icg} \in IC_g$ a program that activates the subcomponents through their control inputs, stores their data outputs into Σ_L, and gives them data inputs taken in Σ_L. These programs may be non-deterministic, and they have a final state. The controller may store the control outputs in its state Σ_C, and all its actions depend on Σ_C.

Definition 3 (controller) *For $\{C_i = (\Sigma_i, \Sigma_i^{init}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$ a set of components, and an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$ for combining them; a controller is a tuple $(\Sigma_C, \Sigma_C^{init} \subseteq \Sigma_C, \Sigma_C^{final} \subseteq \Sigma_C, IC_g \longrightarrow Progs, S)$. A program in Progs is a tuple (μ_S, F) where $\mu_S = (ms_1, ..., ms_n) \in (T_{put} \cup T_{get} \cup T_{act})^n$ is a sequence or micro-steps, such that $T_{put} \subseteq \Sigma_C \times L \times \Sigma_C$ (resp. $T_{get} \subseteq \Sigma_C \times L \times \Sigma_C$) is the set of all possible* put *actions (resp.* get *actions) of the controller, from a state, on a link $\ell \in L$; $T_{act} \subseteq \Sigma_C \times \bigcup_I IC_i \times \Sigma_C$ is the set of all component activations the controller may execute, from a state. $F \subseteq \Sigma_c^{final} \times (OC_g \rightsquigarrow \mathcal{D})$ associates final states of the controller with partial valuations for the global control outputs. $S \subseteq \Sigma_C \times (\bigcup_I OC_i \rightsquigarrow \mathcal{D}) \times \Sigma_C$ defines how the controller stores partial valuations of control outputs of the components into its state.*

5.1.3 Combining Components

Combining components means considering a finite sequence of subcomponents activations and memory storage on the internal links (micro-steps), as a *macro-step* corresponding to a global activation. In order to describe how this is done, we will first describe the micro-steps and how they can be combined into sequences. Then we will define which of these micro-step sequences are the macro-steps of the new component.

In all the section below, we consider an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$ for combining a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$, and a controller $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_C^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs}, S)$ to define their global behavior. We also consider a particular control input $ic_g \in IC_g$, and its associated program (μ_S, F).

5.1.3.1 State of an Assembly

The state of an assembly of components is made of: the state of the controller (an element σ_C of Σ_C), the states of the components (an element σ_I of $\Sigma_I = \prod_I \Sigma_i$), the states of the links (an element σ_L of $\Sigma_L : L \longrightarrow M$, where M is the union of all FIFO types associated with the links), the states of the data ports and the control outputs (an element σ_P of $\Sigma_P : (\bigcup_I ID_i \cup \bigcup_I OD_i \cup \bigcup_I OC_i \cup ID_g \cup OD_g \cup OC_g \longrightarrow \mathcal{D})$). For the sake of simplicity, we assume a unique naming of all ports. We will denote such a global state by a tuple $(\sigma_C, \sigma_I, \sigma_P, \sigma_L)$.

Notice that we need a state of the data ports to express the fact that a component makes some of its outputs available (resp. uses some of its data inputs), but does not copy them onto the links (resp. from the links). The `put` and `get` operations of the FIFOs associated with the links do the job. The method `put` will be represented in the semantics by a function put : $M \times \mathcal{D} \longrightarrow M$ where the assigned value is explicit and put(m, v) is the new value of m after the action `m.put(v)`. Similarly, the method `get` will be represented by a function get : $M \longrightarrow \mathcal{D} \times M$.

5.1.3.2 Micro-steps

For a given ic_g, the micro-steps that correspond to what the controller does with the components and the links are described by the following three rules.

The rule [put] expresses that, if from its state σ_C, the controller puts a value on a link ℓ between ports P_1 and P_2, then the global state evolves with a change in σ_C and σ_L only: the link ℓ receives a new value computed by `put` with the value of its producer port P_1.

$$\frac{(\sigma_C,\ \ell = (P_1, P_2),\ \sigma'_C) \in T_{\text{put}}}{(\sigma_C, \sigma_I, \sigma_P, \sigma_L) \longrightarrow (\sigma'_C, \sigma_I, \sigma_P, \sigma_L[\text{put}(\sigma_L(\ell), \sigma_P(P_1)) \ / \ \ell])} \quad [\text{put}]$$

The rule [get] expresses that, if from its state σ_C, the controller gets the value of link ℓ between ports P_1 and P_2, then the global state evolves with a change in σ_C, σ_P and σ_L: the consumer port P_2 of link ℓ receives the value taken from the link.

$$\frac{(\sigma_C,\ \ell = (P_1, P_2),\ \sigma'_C) \in T_{\text{get}}, (d, m') = \text{get}(\sigma_L(\ell))}{(\sigma_C, \sigma_I, \sigma_P, \sigma_L) \longrightarrow (\sigma'_C, \sigma_I, \sigma_P[d/P_2], \sigma_L[m'/\ell])} \quad [\text{get}]$$

The rule [act] expresses that, if from its state σ_C, the controller activates the component $C_\gamma = (\Sigma_\gamma, \Sigma_\gamma^{\text{init}}, IC_\gamma, OC_\gamma, ID_\gamma, OD_\gamma, \mathcal{B}_\gamma)$ through its control input ic_γ, then the first three fields of the global state evolve. The state of the controller is modified because it stores the control outputs of the component that is activated; the state of the component that is activated is modified; the state of the ports is modified, because some of the output ports of the activated component take new values, and its input ports are reinitialized.

$$\frac{\begin{array}{c}(\sigma_C, ic_\gamma,\ \sigma'_C) \in T_{\text{act}},\\ \exists vod, voc, vid, \sigma'_\gamma \text{ such that } (\sigma_\gamma, vid, vod, voc, \sigma'_\gamma) \in \mathcal{B}_\gamma(ic_\gamma)\\ \text{and } \forall x \in dom(vid).\ vid(x) = \sigma_P(x)\\ (\sigma'_c, voc, \sigma''_C) \in S\\ \sigma'_P = \sigma_P[vod(x)/x][voc(y)/y][\perp/z], \forall x \in dom(vod), \forall y \in dom(voc), \forall z \in ID_\gamma\end{array}}{(\sigma_C, \sigma_I = (\sigma_1, \sigma_2, ...\sigma_\gamma, ..., \sigma_n), \sigma_P, \sigma_L) \longrightarrow (\sigma''_C, \sigma'_I = (\sigma_1, \sigma_2, ...\sigma'_\gamma, ..., \sigma_n), \sigma'_P, \sigma_L)} \quad [\text{act}]$$

The transitions in $\mathcal{B}_\gamma(ic_\gamma)$ that can be taken are those whose input valuation $vid : ID_\gamma \rightsquigarrow \mathcal{D}$ corresponds to what's available in the ports. σ''_C is the modification of σ'_C by storing the values of oc; in σ'_P the input ports of the component C_γ are reinitialized; the output ports in $dom(voc) \cup dom(vod)$ are modified according to the valuations vod and voc of the transition, these are the ports on which the component writes during the transition.

5.1.3.3 Macro-steps

The global component is of the form $C_g = (\Sigma_g, \Sigma_g^{\text{init}}, IC_g, OC_g, ID_g, OD_g, \mathcal{B}_g)$. $\Sigma_g = \Sigma_C \times \Sigma_I \times \Sigma_P$, where Σ_I represents the states of the controller, Σ_I the states of the components, and Σ_P the state of the ports. Notice that the state of the links does not appear here, because the links' values are not persistent across global activations.

The initial configuration $\Sigma_g^{\text{init}} = \Sigma_C^{\text{init}} \times \prod_I \Sigma_I^{\text{init}} \times \sigma_P^{\text{init}}$ is made of the initial state of the controller, the initial states of the components, and some initial state for the ports, that we may leave undefined. Indeed, the initial state is irrelevant because, if the controller is correct[1], then a port is never read before being written to. Hence $\forall p.\sigma_P^{\text{init}}(p) = \bot$.

The behavior $\mathcal{B}_g(ic_g)$ of the composed component for the particular control input ic_g we've been considering so far is a relation $R \subseteq (\Sigma_g \times (ID_g \rightsquigarrow \mathcal{D}) \times (OD_g \rightsquigarrow \mathcal{D}) \times (OC_g \rightsquigarrow \mathcal{D}) \times \Sigma_g)$.

The rule [mac] shows that the tuples of this relation R are deduced from the program of the controller (μ_S, F) associated with the global control input ic_g. The sequences of micro-steps $\mu_S = (ms_0, ms_1, ..., ms_n)$ ends in a final state of the controller. A macro-step only remembers the initial state and the final state of this sequence, the valuations of the data inputs and outputs are deduced from the state of the global ports, and the valuation of the control outputs is given by the values associated with the final state σ_C^n of the controller, via the function F of the program associated with ic_g.

The states of links are not persistent across the activations of the composed component. It means that each macro-step starts with the initial value of the links $\forall l \in L.\sigma_L^0(l) = m$ where m is the new value of m after `m.init` (see section 5.1.2).

$$\frac{\begin{array}{c}(\sigma_C^0, \sigma_I^0, \sigma_P^0, \sigma_L^0) \xrightarrow{ms_0} (\sigma_C^1, \sigma_I^1, \sigma_P^1, \sigma_L^1) \xrightarrow{ms_1} ... \xrightarrow{ms_n} (\sigma_C^n, \sigma_I^n, \sigma_P^n, \sigma_L^n) \\ \text{and } \sigma_C^n \in \Sigma_c^{\text{final}} \text{ and } (\sigma_C^n, voc_g) \in F\end{array}}{((\sigma_C^0, \sigma_I^0, \sigma_P^0),\ \sigma_P^0(ID_g), \sigma_P^n(OD_g), voc_g, (\sigma_C^n, \sigma_I^n, \sigma_P^n)) \quad \in \mathcal{B}_g(ic_g)} \quad \text{[mac]}$$

5.2 42 Control Contracts

5.2.1 Original Form of Control Contracts

The contracts used to describe components in the examples of Section 4.2 are finite-state automata, whose transitions are labeled with various elements, including conditional data dependencies. The variables denoted by Greek letters, used to refer to the values of control outputs, may be used in conditions later on in the contract. The set of such variables is noted $\mathcal{V}$.

Definition 4 (Contracts) *For a component $C = (\Sigma, \Sigma^{init}, IC, OC, ID, OD, \mathcal{B})$, the contract is an automaton $P = (S, S^{init} \subseteq S, \mathcal{V}, IC, OC, ID, OD, T \subseteq S \times LAB \times S)$. LAB is the set of transition labels. A label is a tuple of the form (c, ci, ic, aoc, co, ev) where:*

- *c is a condition on the variables of $\mathcal{V}$, it tells if the transition is possible.*
- *ci describes the conditional inputs required by the component before the activation.*
- *ic is a single control input in IC. It should be used to activate the component.*
- *aoc describes the control outputs provided by the component after the activation, and the way their values are referred to by the variables in $\mathcal{V}$.*
- *co describes the conditional outputs provided by the component after the activation.*
- *ev describes the explicit values associated with some of the ports*

5.2.1.1 Labels Definition

In the sequel, we describe the elements associated with a label of a transition. We define each of them over a label instance (c, ci, ic, aoc, co, ev) and illustrate their relation with the concrete syntax we have been considering so far through the various examples. We take the following label as an example:

[1]The correctness of a controller is defined in section 5.3.2

$$[\alpha = a]\{id_1;\ if\ (\beta = b)\ then\ id_2\}ic_1/\alpha = oc_1\{od_1;\ od_2 = true\}$$

The rule [C] describes the set of activation conditions that may be associated with a transition. An element $c \in C$ is a Boolean function on the values of the variables in $\mathcal{V}$. For $\sigma_v \in \mathcal{V} \longrightarrow \mathcal{D}$, a valuation of the variables in $\mathcal{V}$, if $c(\sigma_v) = true$, the transition is possible, otherwise, it is not possible. In the label example, $c(\{\alpha \mapsto a, \beta \mapsto c\}) = true$, which means that under the configuration where $\alpha = a$ and $\beta = c$ the transition is possible.

$$C = (\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow B \qquad \text{[C]}$$

The rule [CI] describes the set of functions for declaring the conditional inputs. For $\sigma_v \in \mathcal{V} \longrightarrow \mathcal{D}$ a valuation of the variables in V, $ci(\sigma_v)(id)$ tells whether the input id is required or not. In our example, $ci(\{\alpha \mapsto a, \beta \mapsto c\})(id_2) = false$, which means that under the configuration where $\alpha = a$ and $\beta = c$ the input id_2 is not required for the activation of the component with ic_1.

$$CI = (\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow ID \longrightarrow B \qquad \text{[CI]}$$

The rule [AOC] describes the set of possible tuples to declare the control outputs provided by the component and the way their values are referred to by the variables in $\mathcal{V}$. For a given $aoc \subset AOC$ associated with a transition, $(\alpha, oc) \in aoc$ means that the control output oc is produced by the component, and its value is referred to by the variable α. In our example $aoc = \{(\alpha, oc_1)\}$.

$$AOC = \{(\alpha, oc) | \alpha \in \mathcal{V}, oc \in OC\} \qquad \text{[AOC]}$$

The rule [CO] describes the set of functions to declare the conditional outputs for a transition. For $\sigma_v \in \mathcal{V} \longrightarrow \mathcal{D}$ a valuation of the variables in V, $co(\sigma_v)(od)$ tells whether the output od is produced after the activation or not. In our example, there is no condition associated with the outputs `od1` and `od2`, they are produced whatever the valuation of the variables in $\mathcal{V}$.

$$CO = (\mathcal{V} \longrightarrow \mathcal{D}) \longrightarrow OD \longrightarrow B \qquad \text{[CO]}$$

The rule [EV] describes the set of possible partial functions to declare the explicit values associated with the input/output ports of the component. Given a particular function $ev \in EV$, $ev(p) = v$ means that the port p is associated with the value v. If p is an input data port, the component requires the value v on its input p. If p is an output data (resp., control) port, the transition guarantees that the component will produce the value v on the output data (resp., control) port p. In our example, $ev = \{od2 \mapsto true\}$.

$$EV = \{(ID \cup OD \cup OC) \rightsquigarrow \mathcal{D}\} \qquad \text{[EV]}$$

5.2.2 Expanded Form of Control Contracts

Using the variables in $\mathcal{V}$, is a convenient way to write contracts, but if the types of the output control variables are finite, this does not add to the expressiveness of the contract language.

The contract may be expanded in order to make the variables in $\mathcal{V}$ disappear. The new contract is also an automaton where the valuations of $\mathcal{V}$ are added to the states of the contract. Also, we add the explicit values associated with the ports of the component.

A state of the new contract is an element of $SV = S \times (\mathcal{V} \longrightarrow \mathcal{D}) \times ((ID \cup OD \cup OC) \rightsquigarrow \mathcal{D})$. For short, a state will be noted $sv = (s, \sigma_v, ev)$, where s is a state from the original contract,

σ_v is a valuation of the variables in $\mathcal{V}$, and ev is the function used to declare the explicit values associated with the port of the component.

The expanded contract is $PV = (SV, SV^{\text{init}} \subseteq SV, IC, OC, ID, OD, TV \subseteq SV \times \text{LAB'} \times SV)$, but now the labels in LAB' are of the simpler form $(req, ic, ocs, prod)$ where $req \in 2^{ID}$, $ic \in IC$, $ocs \in 2^{OC}$, $prod \in 2^{OD}$ are, respectively, the set of required inputs, the unique input control port, the set of control outputs produced, and the set of data outputs produced.

Definition 5 (Expansion) *The rule [EX] describes the expansion of a general contract p into a simpler contract p′. It gives the transitions of p′ in terms of the transitions of p.*

For a given transition t in the original contract for which the global condition evaluates to `true` *(i.e., $c(\sigma_v) = true$), the set req (resp., prod) contains the set of the required inputs (resp., produced outputs), i.e., those for which the condition function ($ci(\sigma_v)$) (resp., ($co(\sigma_v)$)) returns* `true`*.*

The set ocs contains the set of control outputs that were produced. σ'_v is the new valuation of the variables in $\mathcal{V}$; notice that it generates one transition per value in the type of each control output. To each target state of the generated transitions is associated the function ev of the original transition.

The initial states are defined by: $SV^{init} = S^{init} \times \{\{v \mapsto \perp | v \in \mathcal{V}\}\} \times \{\{\}\}$. Each initial state is composed of an initial state of the original contract, a valuation that maps an undefined value (i.e., $\perp$) to each variable in $\mathcal{V}$, and an empty valuation function of the ports.

$$\frac{\begin{array}{c} t = (s, (c, ci, ic, co, aoc, ev), s') \in T,\ c(\sigma_v) = true, \\ req = \{id \mid ci(\sigma_v)(id)\},\ prod = \{od \mid co(\sigma_v)(od)\},\ ocs = \{oc \mid (\alpha_x, oc) \in aoc\} \\ \sigma'_v = \sigma_v[d_1/\alpha_1, ...d_n/\alpha_n] \text{ where } aoc = \{(\alpha_k, oc_k)\}_{k=1..n},\ \text{and } d_k \in \text{type\_of}(oc_k) \end{array}}{((s, \sigma_v, ev'), (req, ic, ocs, prod), (s', \sigma'_v, ev)) \in TV} \quad [EX]$$

Example Figure 5.1 illustrates the effect of expanding one transition of an original contract `(a)` when $\sigma_v = \{\alpha \mapsto a, \beta \mapsto c\}$, and the type of oc_1 control output is `{a, b}`. The expansion generates two transitions in the expanded contract `(b)` because the variable α is associated with the control output oc_1 which takes its value in `{a, b}`. The variable β keeps the same value because it is not associated with a control output in this transition. The control input id_2 does not appear in the generated transitions because under the configuration described by σ_v, id_2 is not required. To each of the target state of the transitions is associated the function $ev = \{od_2 \mapsto true\}$, because we know that after this activation, this output will take the value `true`.

5.2.3 The Master/Slave Relation

We introduced, informally, the master/slave relation in Section 3.4.2.2. In some sense, this relation implies the activation of two components in one atomic macro-step. This relation cannot be part of the contract of a component, because it involves two distinct components. The master/slave relation is defined over a set of assembled components.

Definition 6 (master/slave) *Given a system defined as an assembly of a set of components $\{C_i = (\Sigma_i, \Sigma_i^{init}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$, the master/slave relation is defined by the function $arch : init \rightarrow tar$ where:*

- *$init \subseteq \bigcup_I C_i \times \bigcup_I OC_i$ is the subset of initiator components, and their corresponding control outputs. These components are those requesting the activation of another component.*

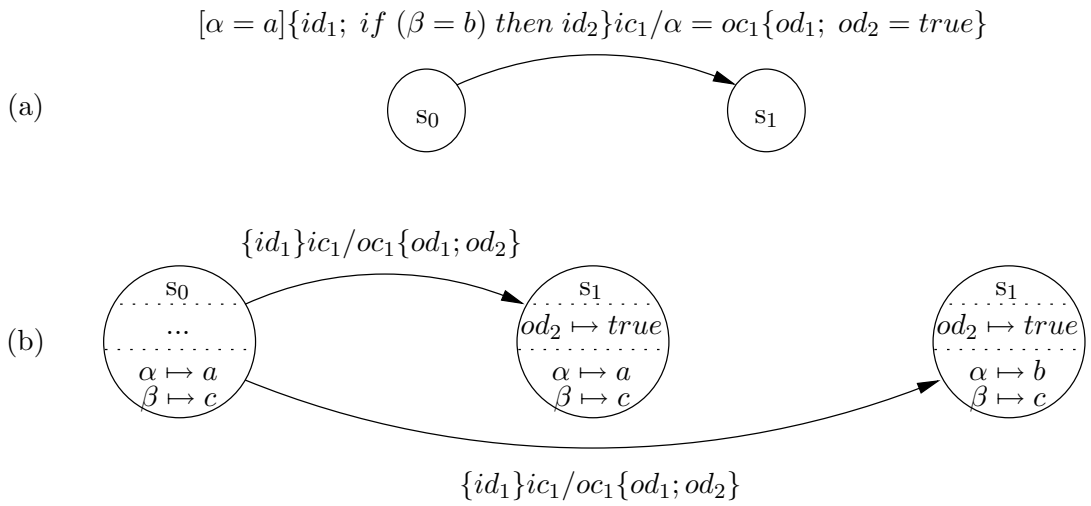


Fig. 5.1: The original (a) and expanded (b) form of a transition when the type of oc_1 is {a, b}

- *$tar \subseteq \bigcup_I C_i \times \bigcup_I IC_i$ is the subset of target components, and their corresponding control inputs. These components are those activated in response to an initiator request.*

5.3 Formal Definition of Consistency

In the sequel, we define the consistency relation between a component and its contract in Section 5.3.1, and the consistency of the controller with the contract of the components in Section 5.3.2. We assume the controller is consistent with the contract when defining the consistency of the component and vis-versa.

5.3.1 Contracts Vs Basic Components

Given a component $C = (\Sigma, \Sigma^{\text{init}}, IC, OC, ID, OD, \mathcal{B})$, and the expanded form of its contract $PV = (SV, SV^{\text{init}} \subseteq SV, IC, OC, ID, OD, TV \subseteq SV \times \text{LAB'} \times SV)$, we can see the component and its contract as two transition systems, each one having its set of states (Σ and SV respectively) and a transition relation ($\mathcal{B}$ and TV respectively).

A contract is an abstraction of the behavior of a component. A state of the contract corresponds to some states of the component, and each transition in the contract describes the behavior of the component during an activation. That is, what the inputs (resp., outputs) used (resp., produced) by the component are, and possibly their values.

The consistency between a component and its contract is defined by means of a simulation relation (see the rule [SC] below). The alphabets of the two transition systems are disjoint: a set of labels for the contract, and a set of behaviors for a component. Hence, to define the usual simulation relation, we need to define a relation between a label of a contract and a behavior of a component, first. This relation is described by the rule [AC].

Activation Consistency The rule [AC] describes the relation $AC \subseteq \mathcal{B} \times LAB'$. The behavior associated with an activation $b \in \mathcal{B}(ic)$ is consistent with the label $l = (req, ic, ocs, prod)$, if the activation of the component with ic:

- uses not more than the inputs declared in the label ($dom(vid) \subseteq req$), produces at least the data and control outputs as declared in the label ($prod \subseteq dom(vod)$ and $ocs \subseteq dom(voc)$

respectively).
- associates the same explicit values with the data and control ports as described by the function ev'

$$\frac{\begin{array}{c} l = (req, ic, ocs, prod).\ ((s, \sigma_v, ev), l, (s', \sigma'_v, ev')) \in TV \\ b = (\sigma, vid, vod, voc, \sigma').\quad b \in \mathcal{B}(ic) \\ dom(vid) \subseteq req,\ \ prod \subseteq dom(vod),\ \ ocs \subseteq dom(voc) \\ \forall id \in (ID \cap dom(ev'))vid(id) = ev'(id) \\ \forall od \in (OD \cap dom(ev'))vod(od) = ev'(od) \\ \forall oc \in (OC \cap dom(ev'))voc(oc) = ev'(oc) \end{array}}{(b, l) \in AC} \quad [\text{AC}]$$

The Simulation Relation The rule [SC] says that a state s of the contract simulates a state σ of the component if and only if for all behaviors b corresponding to an activation of the component with a control input ic, there exists a transition labeled with l in the contract, which is consistent with the behavior of the component (i.e., $(b, l) \in AC$), and leads the contract to a state s' that simulates the new state of the component after the activation (i.e., $(s', \sigma') \in SC$).

$$\begin{array}{l} \forall (s, \sigma) \in S \times \Sigma \\ \forall (ic, b) \in \mathcal{B}.\quad b = (\sigma, vid, vod, voc, \sigma') \\ (s, \sigma) \in SC \end{array} \implies \begin{array}{l} \exists l \in LAB'.\quad l = (c, ci, ic, aoc, co, ev), \\ (b, l) \in AC,\quad s \xrightarrow{l} s' \\ (s', \sigma') \in SC \end{array} \quad [\text{SC}]$$

Definition 7 (Consistency of a Component with its Contract)
The rule [CC] describes the consistency relation between a component and its contract. It says that a component $C = (\Sigma, \Sigma^{init}, IC, OC, ID, OD, \mathcal{B})$ *is consistent with its contract* $PV = (SV, SV^{init} \subseteq SV, IC, OC, ID, OD, TV \subseteq SV \times LAB' \times SV)$ *if and only if each initial state of the component is simulated by an initial state of its contract.*

$$\forall \sigma \in \Sigma^{init},\quad \exists s \in SV^{init}.\quad (s, \sigma) \in SC \qquad [CC]$$

5.3.2 Contracts Vs Controllers

In the sequel, we consider:

- A set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$
- The set $\{PV_i = (SV_i, SV_i^{\text{init}} \subseteq SV_i, IC_i, OC_i, ID_i, OD_i, TV_i \subseteq SV_i \times \text{LAB'}_i \times SV_i)\}_I$ of their expanded contracts.
- An architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$
- A controller $(\Sigma_C, \Sigma_C^{\text{init}} \subseteq \Sigma_C, \Sigma_c^{\text{final}} \subseteq \Sigma_C, IC_g \longrightarrow \text{Progs}, F, S)$

The consistency of the controller expresses that, given the way the components are assembled with the architecture, and given their contracts, the controller is *correct*, i.e., it uses the components according to their contracts.

Since the controller activity (e.g., `ctrl` of Figure 5.2) happens only in response to the activations of the global component (e.g., MAIN of Figure 5.2), the consistency relation requires that the controller respects the contract PV_i (e.g., Figure 5.3) of each component C_i (e.g., the component B) *for all sequences of global activations allowed by the contract of the global component.* In the sequel, we assume that the global component is activated correctly according to its contract.

We can check the consistency of the controller with respect to each component contract separately. We note $T_{act_i} \subseteq (\Sigma_C \times IC_i \times \Sigma_C)$ the set of activations of the component C_i;

$T_{get_i} \subseteq (\Sigma_C \times L_i \times \Sigma_C)$ (where $L_i = \{(a,b) \in L.b \in ID_i\}$), the set of `get` operations on the wires connected to one of the input ports of C_i.

For a sequence of global activations Ω_S, the controller outputs a sequence of ordered micro-steps $\mu_S = (ms_0, ..., ms_n) \in (T_{act} \cup T_{put} \cup T_{get})^n$. In order to capture only the activations of C_i in the sequence of micro-steps μ_S, we denote $\mu_S/T_{act_i} = (ms_0, ..., ms_k) \in (T_{act_i})^k$ where $k \leq n$ the sequence obtained from μ_S by removing all the micro-steps but the activations of the component C_i. Similarly, $\mu_S/(T_{act_i} \cup T_{get_i})$ captures its activations and data input assignments.

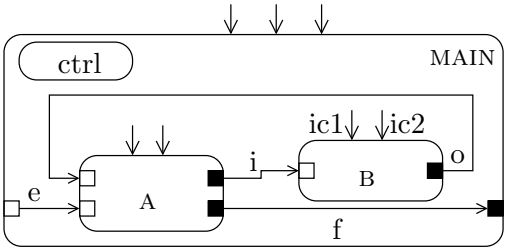


Fig. 5.2: An assembly of components

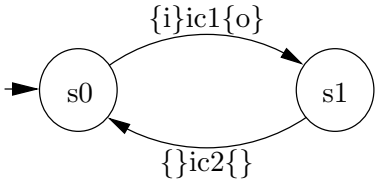


Fig. 5.3: A contract for the component B

The definition of the consistency relation can be split into two sub-properties, that should be both true:

First, each contract PV_i can be seen as a recognizer of the language of correct activation sequences of C_i; we can just erase anything in the labels of PV_i but the control inputs, and we get a *language recognizer* in the form of a non-deterministic finite automaton $A_i = (SA_i = SV_i, SA_i^{\text{init}} = SV_i^{\text{init}}, SA_i^f = SA_i, IC_i, TA_i \subseteq SA_i \times IC_i \times SA_i)$, on which all states have to be considered as accepting states. The rule [rec1] defines the transitions set of A_i. Figure 5.4 illustrates the automata associated with the contract of the component B. The words to be tested are obtained by keeping only the activations of C_i (i.e., μ_S/T_{act_i}). For each possible Ω_S, the corresponding μ_S/T_{act_i} has to be accepted by A_i, for all C_i (i.e., $\mu_S/T_{act_i} \in \mathcal{L}(A_i)$).

$$\frac{(s, (req, ic, ocs, prod), s') \in TV_i}{(s, ic, s') \in TA_i} \quad \text{[rec1]}$$

Second, for each activation of a component, the controller should provide the required inputs. The operation `get` on the FIFO associated with the wire connected to an input port is considered as an assignment to this port. In the same sense of A_i we construct an automaton $D_i = (SD_i = SV_i \times Av, SD_i^{\text{init}} = SV_i^{\text{init}} \times \{(ID_i \rightarrow false)\}, SD_i^f = SD_i, ID_i, IC_i, TD_i \subseteq SD_i \times IC_i \cup ID_i \times SD_i)$. The states of D_i are associated with a total function in the set $Av \subseteq (ID_i \rightarrow B)$. It defines whether an input has been assigned a new value or not (i.e., if it is available). The initial states are associated with the function returning `false` because no input is supposed to be available. The set of transitions TD_i is constructed following the rules [rec2] and [rec3].

$$\frac{\begin{array}{l}(s, (req, ic, ocs, prod), s') \in TV_i \\ \forall id \in req,\ av(id) = true \\ \forall id \in req,\ av' = av[id/false]\end{array}}{((s, av), ic, (s', av')) \in TD_i} \quad \text{[rec2]}$$

$$\frac{\begin{array}{l}(s, av) \in SD_i \\ \exists id \in ID_i,\ av' = av[id/true]\end{array}}{((s, av), id, (s, av') \in TD_i} \quad \text{[rec3]}$$

Figure 5.5 illustrates such an automaton for B. The rule [rec2], tells that the activation of the component is only possible in the states where all the required data inputs are available. After the activation, the required inputs are no longer available. The rule [rec3], defines the

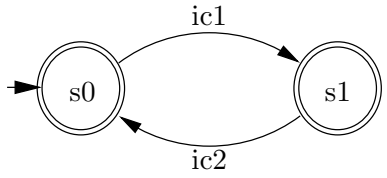


Fig. 5.4: Activations recognizer for B

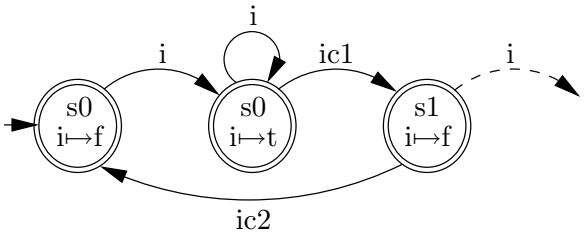


Fig. 5.5: Data dependencies for B

transitions corresponding to the data input assignments. When id (i in Figure 5.5) is assigned a new value, we move to a state where $av'(id) = true$.

The controller respects the data dependencies of the component C_i, if it provides the component with the required input before it activates it. In other words, the controller is consistent with the contract if the sequence $\mu_S/(T_{act_i} \cup T_{get_i})$ is recognized by D_i. Notice that $\mu_S/(T_{act_i} \cup T_{get_i}) \in \mathcal{L}(D_i) \implies \mu_S/T_{act_i} \in \mathcal{L}(A_i)$. We introduced A_i to a better understanding of the concept.

Chapter 6

Exploiting 42 Control Contracts

Introduction (En) *The control contracts we adopted for 42 are basically used for the specification of components in order to use them correctly. This chapter presents some interesting uses of the contracts. A 42 control contract relates a component to the controller activating it; the consistency issues between controllers/components and the contracts are presented at first. At a second place, we show how controllers may be deduced from the contracts. We present static code generation for 42 controllers describing the synchronous* MoCC*, and contract interpreters for the asynchronous* MoCC*.*

Contents

Introduction (Fr) *Les contrats de contrôle que nous avons adopté pour le modèle 42 sont essentiellement dédiés à la spécification des composants afin de les utiliser correctement. Dans ce chapitre, nous introduisons d'autres usages intéressants de ces contrats. Dans un premier temps, nous montrons l'utilité des contrats pour observer différentes notions de compatibilité entre les composants/contrôleurs et les contrats. Ensuite, nous montrerons comment les contrats peuvent être utilisés pour déduire le code d'un contrôleur : nous donnons une exemple de génération de code de contrôleur pour des assemblages synchrones décrits en 42, ainsi que la technique d'interprétation de contrats pour déduire des contrôleurs pour le* MoCC *asynchrone.*

6.1 Contracts and Consistency Checking

In design by contracts (see Section 2.3.3), a contract holds between two or more entities of a system. The point of view of design by contracts community is to avoid *defensive programming* to check the integrity of a contract. Instead, checking the contract should be a feature of the design framework. This helps separate the implementation from its specification.

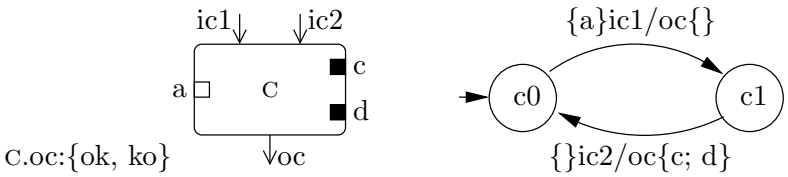


Fig. 6.1: A component and its contract

In the case of 42, the contract holds between a component and the controller activating the component. The consistency properties we are interested in was introduced informally in Section 3.3; Section 5.3 deals with their formalization. We require that:

- The component must be a good implementation of the contract associated with it.
- The controller must activate the component respecting the constraints expressed by its contract.

In the sequel we illustrate over the example of Figure 6.1, how do we check the consistency of the components/controllers with the contracts. Notice that we consider the expanded form of control contracts as described in Section 5.2.2. We assume the controller to be consistent with the contract when checking the component implementation and vice-versa.

Figure 6.1 is an example component and its contract. The interface of the component exposes two input control ports to activate it (`ic1, ic2`), one input and two output data ports (`a` and `c, d`) respectively, and one output control port `oc` which takes its values in {`ok, ko`}. The behavior of the component as described by the contract is as follows: first, the component should be activated with `ic1`, this activation requires a value on the input `a` and produces the value of the control output `oc`. Second, it should be activated with `ic2` which requires no inputs and provides outputs on `c` and `d` data ports.

6.1.1 Checking Component Implementation

The consistency of a component with its contract may be checked dynamically, considering the contract as a monitor evolving in parallel with the component. The consistency checking consists in the verification that: for each activation, the behavior of the component corresponds to what is expected by the contract. That is to say, the component should not use an input that is not declared as required in the corresponding transition, and should produce the values of the data and control outputs of the component as described by the contract.

To check the consistency, we need to know whether the component accessed a port of its interface or not. A way to do it, dynamically, is to allow port access using accessors only (Figure 6.2). Each port is equipped with a Boolean value which indicates whether the port has been accessed or not. Before the activation, the Boolean value of each port is initialized to false through *init()*. After the activation with the control input `ic`, we collect all the used (resp., produced) input (resp., output) data ports in the set $Data_U^{ic}$ (resp., $Data_P^{ic}$). Also the control ports are collected in the set $Control_P^{ic}$. We compare these sets with the set of required data inputs, provided data outputs and the provided control outputs corresponding to the transition that has been taken.

In the example illustrated above, we check that: $Data_U^{ic1} \subseteq \{a\}$, $\{c, d\} \subseteq Data_P^{ic1}$, $\{oc\} \subseteq Control_P^{ic2}$. That is, to check that: when activated with `ic1`, the component didn't use more than the input `a`; and produced at least the outputs `c, d`; its activation with `ic2` should produce at least the value of the control output `oc`. If one of these inclusion tests fails, the component is not consistent with its contract.

Our contracts are not limited to express the input output data dependencies. The transitions

```
class InputPort<Type>{
  Type value;
  bool used;
  init(){ used=false;}

  Type read(){
    used = true;
    return value;
  }
}
```

```
class OutputPort<Type>{
  Type value;
  bool produced;
  init(){ produced=false;}

  write(Type val){
    produced=true;
    value=val;
  }
}
```

Fig. 6.2: Accessors to port values

may be decorated to expose the values expected on the output ports. Following the same reasoning, this example may be extended to check if the component puts the right value on an output port as described by the contract.

6.1.2 Checking Controller Micro-steps

The consistency relation that should hold between the controller and the contracts of the components was addressed formally in Section 5.3.2. We showed that the consistency checking may be performed locally to a component. The only thing we need to know is if the controller activates the component respecting the control sequences described by the contract; and for each activation, if the component is provided with its required inputs. The formal description led us to the definition of a recognizer of micro-steps sequences. The automaton describes how the component should be activated, and how it should be provided with the required inputs.

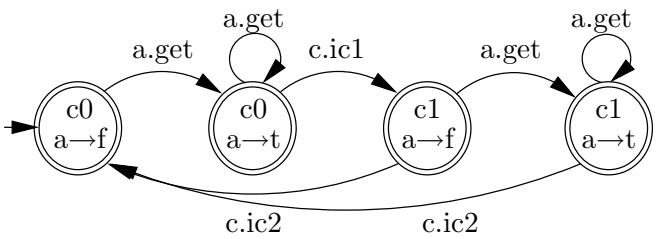


Fig. 6.3: A recognizer of the correct micro-steps for the component in Figure 6.1

Figure 6.3 illustrates a recognizer of the correct micro-steps sequences extracted from the contract of Figure 6.1. We recall that each state is associated with a function that tells whether the input `a` is provided to the component or not. `a.get` micro-step makes the input available, `c.ic1` indicates that the input has been used.

To check the consistency of the controller with the contracts of the components. We extract the recognizers from their respective contracts (their alphabet is disjoint). The recognizers are the basis of monitors that are instantiated with each component. The micro-steps corresponding to the component C makes its recognizer evolve. If the micro-step does not match a transition in the recognizer, the controller is not consistent with the contract of C.

6.2 Deducing the Controller from the Contracts

So far, we presented several modeling examples with 42. The controllers associated with the assemblies were written in an imperative style language. Instead of writing the controllers and

checking their consistency with the contracts later (see Section 6.1.2), we propose in this section to generate them automatically. Automatic generation of the controllers depends on the *MoCC* and relies on the information provided by the contracts and the architecture of the assembly. It guarantees the controllers to be consistent with the contracts.

In Section 6.2.1 we describe static code generation which yield a controller written in an imperative style language similar to what we have presented so far. Section 6.2.2 describes contract interpreters where the controller micro-steps are generated at runtime. The distinction between the two approaches is what would distinguish static scheduling of activities from their dynamic scheduling.

6.2.1 Static Code Generation for Synchronous Controllers

This section describes how contracts may help in generating the code of a controller for the *synchronous MoCC*. For such a *MoCC*, we know that each global `geto` (resp., `go`) consists of a sequence of activations of the components with `geto` (resp., `go`). The architecture and the contracts describe the data dependencies.

Generating the code of the controller consists in the computation of a partial order of the controller micro-steps. First, the partial order is used to detect the presence of cyclic data dependencies. Second, if there is no cycles, a total order is extracted from the partial order.

6.2.1.1 Computing the Partial Order of the Controller Micro-Steps

Let us consider a set of synchronous components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$ and an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$ for combining them. We note $P_c = (S_c, S_c^{\text{init}} \subseteq S_c, \mathcal{V}_c, IC_c, OC_c, ID_c, OD_c, T_c \subseteq S_c \times \text{LAB}_c \times S_c)$ the contract of the component c. A transition of a contract is noted $t = (s, (req, ic, ocs, prod), s')$. We note $t \in T_c$ a transition in the contract of the component c.

Below, the rules [ARCH] and [CONT] describe how the partial order of the micro steps is computed for a global activation with `geto`.

The rule [ARCH] relies on the architecture of the assembly. It tells that a `put` operation on a wire l always precedes the operation `get` on the same wire. The precedence is noted $l.put \prec l.get$

The rule [CONT] relies on the information about the data dependencies of the outputs. It says that for a given transition of the contract associated with the computation of some outputs of the component c (i.e., $\texttt{geto}_x$), the operation `get` on a wire connected to a required input (i.e., l_i) precedes the activation of the component (i.e., $\texttt{c.geto}_x$), which in turn precedes the operation `put` on the wires connected to the provided outputs (i.e., l_o).

$$\frac{l = (p1, p2) \in L}{l.put \prec l.get} \quad \text{[ARCH]} \qquad \frac{\begin{array}{c} t = (s, (req, geto_x, ocs, prod), s') \in T_c \\ \forall l_i = (p, id) \in L.id \in req \\ \forall l_o = (od, p) \in L.od \in prod \end{array}}{\begin{array}{c} l_i.get \prec c.geto_x \\ c.geto_x \prec l_o.put \end{array}} \quad \text{[CONT]}$$

6.2.1.2 Complete Vs Partial Computation of the Outputs

From the partial order of micro-steps, one computes a total order consistent with it using topological sorting algorithms. Such a complete order corresponds to the code associated with a global control input (implemented by the controller). However, it does not allow for partial computation of the outputs.

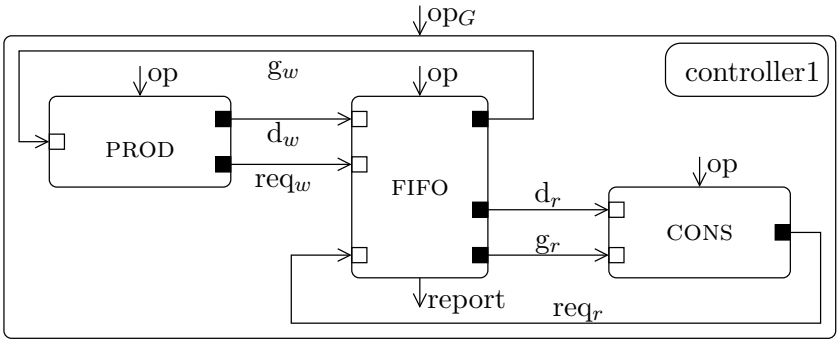
For partial computation of the outputs, we extract a partially ordered set from the initial set for each output. The extracted set for the computation of a given output $\texttt{o}_x$ consists of a chain that contains `l.get` as a minimal element, where the micro-step `l.get` defines the value of the output $\texttt{o}_x$. The maximal elements of that chain correspond to activations of `pre` components, or to micro-steps of the form `d.put`; these micro-steps corresponds to reading from global data inputs (the dependencies of the output $\texttt{o}_x$). Then, we compute a complete order on each chain. Each complete order is associated with a particular `geto` of the controller.

6.2.2 Asynchronous Simulation Controllers as Contracts Interpreters

This section deals with the deduction of the code of a controller for an asynchronous *MoCC*. Contrary to the static code generation described previously in Section 6.2.1, the micro-steps associated with each macro-step are computed on the fly.

Figure 6.4 recalls the architecture of a simple producer/consumer system. This example is described in Section 4.2.3; there, we associated controllers written in an imperative-style. We commented on the difficulty of writing controllers with fine grained simulation steps.

The controllers we describe in this section are based on *contract interpretation*. They allow for the simulation of the system at a granularity defined by the transitions of the contract of each component. We first describe how the system may be simulated by interpreting its contract alone, then we comment on the simulation of contracts plus the implementation in Section 6.2.2.3.



Ports Types :

$d_r, d_w : int$	$g_w, r_r : \{t, f\}$ (true, false)
FIFO.$report : \{ok, ko\}$	$req_r, req_w : \{t\}$

Fig. 6.4: The producer/consumer example in 42

6.2.2.1 Contract Interpretation Principle

In 42 simulation models like the one of Figure 6.4, the atomicity is expressed by the `op` control input. Each activation of the component with the input `op` makes it execute a (potentially long but terminating) piece of its behavior.

In a 42 model made of two components `C1` and `C2`, we can consider each of the components to be an automaton, whose transitions are labeled by `op`. The simulation produces the paths of the asynchronous product of these two automata (see Section 2.1.2).

Instead of modeling the behavior of components with general automata, we consider their contracts. The contracts define an abstract view of the behavior of the components. They contain all the information needed to understand the explicit synchronizations between the

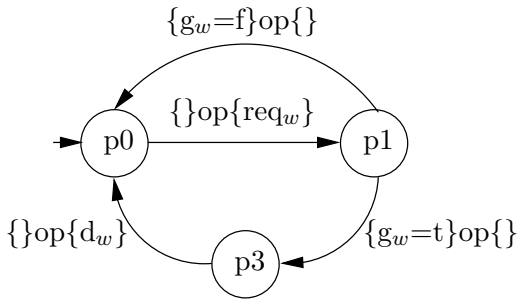


Fig. 6.5: Contract of the producer

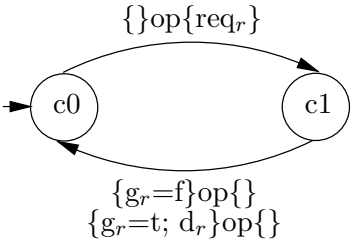


Fig. 6.6: Contract of the consumer

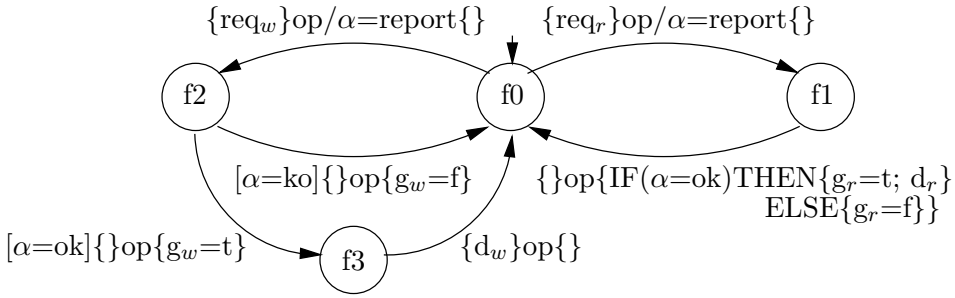


Fig. 6.7: Contract of the FIFO

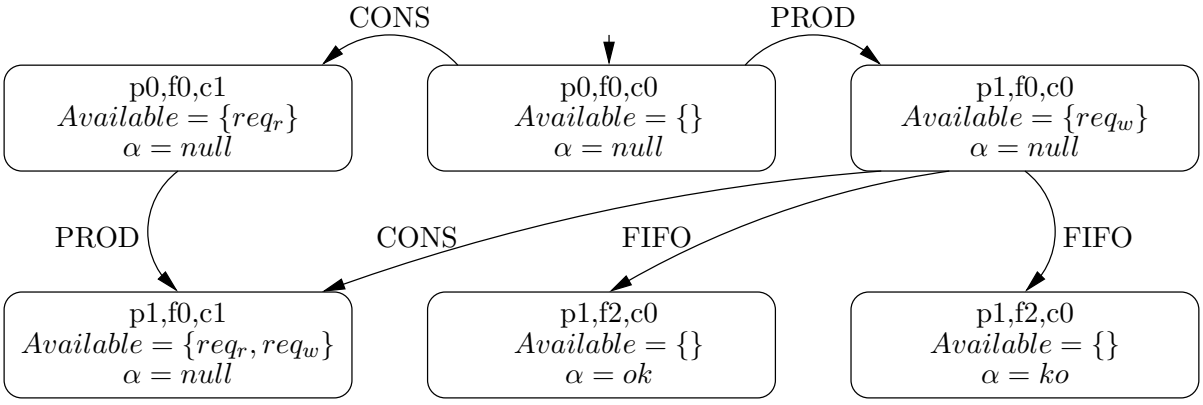


Fig. 6.8: Interleaving component activations

components.

The simulation *MoCC* — and the corresponding 42 controller — can be obtained automatically by interpreting contracts.

Example Let us illustrate all this with the example. The contracts of the producer, the consumer, and the FIFO, are given by Figures 6.5, 6.6 and 6.7 respectively. The types of the ports are given in Figure 6.4. Figure 6.8 gives a small part of the graph whose paths are explored by the non-deterministic contract interpreter. The simulation starts with all the component contracts in their initial state. At each simulation step, the choice of the component to consider is non-deterministic, among the components whose contract shows that at least one transition is possible (all data required are present in the set *Available*). Activating a component consists of choosing a transition in its contract, from the current state. After the activation, the required data are removed from the set *Available*, the provided ones are added to it. The control outputs are given non-deterministic values (in their finite domain).

```
1  Controller interpreter is{
2  CT ⊂ (⋃C_i × ⋃T_i) //Components and their transitions
3  Available ⊂ ⋃ID_i // set of available input data
4  CS ∈ ΠS_i // Contracts states
5  for op_G do{
6       let TR = {t | t = (s, lab, s') ∈ ⋃T_i.s ∈ CS}
7       let ET = Ex^TR_Available
8       let (c,t) ∈ CT.t = (s, lab, s') ∈ ET
9       forall id in req_t {id.put; id.get;}
10      c.ic;
11      Available := (Available\req_t)∪(prod_t)
12      CS := CS\{s} ∩ {s'}
13 }}
```

Fig. 6.9: Sketch of the contracts interpreter

6.2.2.2 Controller Implementation

For a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$ and their respective contracts $\{P_i = (S_i, S_i^0 \subseteq S_i, IC_i, OC_i, ID_i, OD_i, T_i \subseteq S_i \times \text{LAB}_i \times S_i)\}_I$; Figure 6.9 is a sketch of the interpreter controller. For the sake of simplicity, we consider contracts with simple transitions where all the control variables do not appear. This amounts to contract expansion as defined in Section 5.2.2 and does not affect the complexity of the controller algorithm. A transition label in LAB_i will be considered as a tuple $(req \in 2^{ID_i}, ic \in IC_i, ocs \in 2^{OC_i}, prod \in 2^{OD_i})$, where ID_i, OD_i, IC_i, OC_i are respectively the sets of data input, data output, control input and control output ports of the component C_i. In the sequel we adopt the notations req_t, ocs_t, $prod_t$, ic_t for the sets $req, ocs, prod$ and the input control port associated with the label of a transition t.

Computing the Set of Executable Transitions At each simulation step, TR denotes the set of the outgoing transitions. We note $Ex^{TR}_{Available}$, the subset of TR including only the executable transitions when the available inputs are those in the set $Available$. Ex is a function $Ex : (2^{\bigcup T_i} \times 2^{\bigcup ID_i}) \rightarrow 2^{\bigcup T_i}$ defined by the rule [EX]. The rule says that, given a set of transitions TR, and a set of available inputs $Available$, a transition t from TR is executable if and only if its required inputs req_t are included in the set $Available$.

$$\frac{TR \in 2^{\bigcup T_i}, Available \in 2^{\bigcup ID_i}, t \in TR}{(t \in Ex^{TR}_{Available}) \implies req_t \subseteq Available} \quad \text{[EX]}$$

The Interpretation Algorithm The interpreter of Figure 6.9 needs to know about the subcomponents and their possible transitions (line 2). It maintains the set of available inputs for each component (line 3) and the state of each contract (line 4). For each global activation (i.e., a simulation step), the controller computes the set TR of all outgoing transitions from the state of the contracts CS (line 6). From the set TR it computes the set of executable transitions ET using the function $Ex^{TR}_{Available}$ (line 7). It selects randomly one executable transition t among the set ET (line 8). The interpreter provides the component, to which belongs the selected transition with its required inputs (line 9) and activates it with the corresponding control input (line 10). At the end of the macro-step, the controller removes the used inputs from the set $Available$, and adds the provided ones (line 11). It also updates the state of the contracts (line 12).

Contracts' Variables Are Given Non-deterministic Values Notice that the *variables* used to store control outputs values do not appear in the controller because of the use of expanded form of control contracts. The variables are now associated with the states of the contracts (see Section 5.2.2). We recall that each transition in the original contract that may update the value of a variable corresponds to a set of transitions with the same label, leading to distinct states. The target states are distinguished by the values associated with the updated variables. The random selection of transitions in the expanded contract, makes the values assigned to variables non-deterministic.

6.2.2.3 Comments

Executing contracts allow for abstract simulation. Contracts are considered as abstract implementation of components. Executing contracts allows for observing the behavior of the assembly, and checking component synchronizations. This may be done early in the design flow, before the concrete implementation is written.

The granularity of the simulation is defined by the contracts. For each global activation of the system, the controller makes one component execute one transition of its contract.

The concrete implementation may be used when interpreting contracts. The same controller may be used with the concrete implementation of the components. The controller uses the contracts to take the decision to activate the components. The values of the control outputs are no longer non-deterministic, they take the values that are indeed produced by the component. In this case, the simulation exposes exactly the same behavior as in the simulation model described in Section 4.2.3.4. Contract interpreters avoid the complicated and error-prone task of writing the controllers manually.

6.2.2.4 Hierarchic Contract Interpreters

In this section we describe how composed components may be used in an asynchronous model where the simulation is based on contract interpretation. This requires the components be equipped with their specification. Specifying composed components may be done in two ways:

- **With explicit specifications:** one can try to write the control contract of a composed component by hand, mostly inspired by the subcomponent contracts, and the architecture of the assembly. The contract may be deduced automatically but this may yield a huge automaton.
- **Using introspection mechanism:** that is to keep only the contracts of the components at the leaves of the hierarchy, and to compute the contracts of the composed ones on the fly during the simulation. The component should provide additional control ports dedicated to introspection mechanism as described in Section 3.2.3. Hierarchic contract interpreters deal with this option; these are described in the sequel.

Example Consider the system illustrated in the Figure 6.10. The component MAIN is composed of two components P and FC. For each of its subcomponents, we also describe their internal structure. P, only encapsulates a producer component PROD. FC is made of FIFO and CONS components. In this example, we have two levels of hierarchy; the components PROD, FIFO, CONS are at the leaves of the hierarchy. These are described in the system of Figure 6.4.

To simulate the system, one may flatten the architecture, considering only the components at the leaves, put a contract interpreter as the one described in Figure 6.9, and simulate. This approach may be interesting in case we want to gain efficiency, but implies re-engineering the

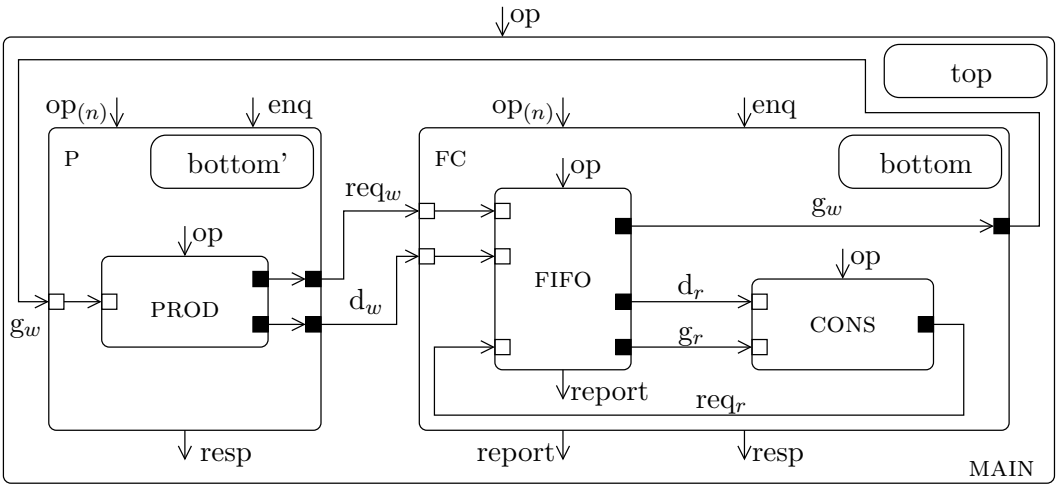


Fig. 6.10: The producer/consumer example in 42

system and assumes we have access to the detailed implementations of the components P and FC.

Basic Idea The approach described in this section allows us to keep the architecture as it is given. It splits the task of the controller interpreter of Figure 6.9 among the controllers at each level of the hierarchy (`top`, `bottom`, and `bottom'` controllers in Figure 6.10). The idea is that:

- Each controller (except `top`) constructs on the fly, the contract of the component encapsulating it. `bottom` (resp., `bottom'`) constructs the contract of FC (resp., P) depending on the contracts of FIFO and CONS (resp., PROD). There is a need for communication between the controllers `bottom` (resp., `bottom'`) and `top`. `top` asks the controllers `bottom, bottom'` for the possible transitions through `enq` (for enquire) input control port. `bottom, bottom'`, respond with a set of transitions through `resp` output control ports.
- Each controller manages the available inputs that depend on the internal connections and let the controller above it manage the ones depending on the global inputs. The controller `bottom` manages the availability of $\mathtt{d}_r$, $\mathtt{g}_r$ and $\mathtt{req}_r$ and let the controller `top` mange $\mathtt{d}_w$, and $\mathtt{req}_w$. The controller `bottom'` does not manage any input.
- The transition to be taken is selected by the controller at the top of the hierarchy. It corresponds to a set of activations that go in depth until reaching a component at the leaves. For example, `top` selects a transition corresponding to the component FIFO; it activates FC through the parameterized $\mathtt{op}_{(n)}$, the parameter corresponds to the index of the transition to be taken (see below); then the controller `bottom` activates FIFO and the step is finished.

What Kind of Specification is Exchanged? The data type for the exchange between the controllers `top` and `bottom` or `bottom'` is a set of indexed transitions $(n, t) \in (N \times \bigcup_i Ti)$ where the index n is a unique identifier of the transition t. The controller `top` does not need to know all the data dependencies of the transition, only those managed by itself are interesting (i.e., $\mathtt{req}_w$, $\mathtt{d}_w$ and $\mathtt{g}_w$). That's why, `bottom` will only communicate a transition where all its local dependencies (i.e., $\mathtt{d}_r$, $\mathtt{g}_r$ and $\mathtt{req}_r$) are hidden using the function $Proj$.

$Proj : \bigcup T_i \times 2^P \rightarrow \bigcup T_i$ where P is the set of all possible component ports; is a function that constructs a new transition from an original one, hiding some port names in its label. The rule

[PROJ] defines it, it says that: given a contract transition t, and a set of ports names p; $Proj_p^t$ is a transition where all ports names are removed except those included in the set p.

$$\frac{\begin{array}{c} t = (s, (req, ic, ocs, prod), s') \in \bigcup T_i \\ p \in 2^P.P = \bigcup ID_i \cup OD_i \cup IC_i \cup OC_i \end{array}}{Proj_p^t = (s, (req \cap p, ic \cap p, ocs \cap p, prod \cap p), s')} \quad \text{[PROJ]}$$

6.2.2.5 Controllers Description

In what follows we describe the controllers needed for the design of hierarchic simulation models, where only the components at the bottom of the hierarchy are equipped with explicit contracts. We start with the controllers that are at the *bottom* of the hierarchy, then the one that is at the *top*. Finally, we will describe controllers that may be used at the *intermediate* levels, when there are more than two levels of hierarchy.

Each controller manages a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, IC_i, OC_i, ID_i, OD_i, \mathcal{B}_i)\}_I$, assembled with an architecture $A = (IC_g, OC_g, ID_g, OD_g, L)$. For the sake of simplicity, each wire and the ports connected to it have the same name. In the case where this does not hold, one can perform naming substitution.

For each controller, the set G describes the set of global ports of the encapsulating component; I defines the set of subcomponents inputs that do not depend on global inputs; NTC is the set of indexed subcomponent transitions that are possible from their actual state.

The Controller at the Bottom is illustrated in Figure 6.11. It knows about its subcomponents and their explicit contract transitions (CT), together with the actual states of the contracts (CS). The local inputs are those described by the set I, their availability is managed through the set $Available$. The global outputs are described by G. NTC is the set of index transitions, and the component to which each transition belongs.

When the component is activated with `enq`, the controller computes the set NTC (line 7); each tuple in this set is composed of a unique integer identifier n, a component c, and a possible transition t from the actual state (s) of the corresponding contract. At line 8, it computes the set TR_I; it is the set of transitions from NTC projected on the local inputs. TR_I is used to resolve local dependencies at line 9. Each tuple in the set ret consists of a projected transition on the global ports $Proj_G^t$ and its index; we keep only the transitions that have their local dependencies resolved ($Proj_I^t \in Ex_{Available}^{TR_I}$). At line 10, the controller assigns the set ret to the control output $resp$.

```
1  Controller bottom is{
2  CT ⊂ (⋃C_i × ⋃T_i); Available ⊂ ⋃ID_i;  CS ∈ ΠS_i
3  NTC ⊂ (N × ⋃T_i × ⋃C_i)
4  G = ID_G ∪ OD_G ∪ IC_G ∪ OC_G;  I = ⋃_i ID_i\G
5  for enq do{
6       NTC = {(n,t,c) | ∃!n ∈ N, ∃(c, t = (s, lab, s')) ∈ CT.s ∈ CS}
7       TR_I = {t_I | ∃(n,t,c) ∈ NTC.t_I = Proj_I^t}
8       ret = {(n,t') | (n,t,c) ∈ NTC.t' = Proj_G^t ∧ Proj_I^t ∈ Ex_Available^TR_I}
9       resp := ret;
10 }
11 for op_(n) do{
12      let t = (s, lab, s').∃(n,t,c) ∈ NTC
13      forall id in req_t {id.put; id.get;}
14      c.ic;
15      forall od in (prod_t\I) {id.put; id.get;}
16      forall oc in (ocs_t ∩ G) {oc := c.oc;}
17      Available := (Available\req_t)∪(prod_t)
18      CS := CS\{s} ∩ {s'}
19 }}
```

Fig. 6.11: Sketch of controller at the bottom of the hierarchy

When activated with $\texttt{op}_{(n)}$, the controller receives the index of the selected transition n. It retrieves the transition t, and the component c from NTC (line 13), provides the component with the required inputs (line 14), activates it (line 15), assigns the global outputs with the corresponding values produced by the component (line 16), copies the control output produced (line 17), finally, it updates the available inputs and the state of the contracts.

The Controller at the Top (Figure 6.12) manages the activation of its subcomponents in the set C, and the set of available inputs $Available$. For each activation with op, the controller constructs the set NTC: it asks the subcomponents for their possible transitions through enq (line 6); the set NTC will contain the set of indexed transitions and their corresponding component. The indexed transitions are gathered from the output control ports $resp$ of each component (line 7). From the set of all transitions TR, it selects randomly one transition t that has it data dependencies resolved ($Ex^{TR}_{Available}$ at line 9). It provides the component to which belongs the transition with the required inputs (line 10), activates it with the corresponding input control port together with the index of the transition (line 11). After the activation it updates the set of available inputs (line 12).

6.2.2.6 Extending the Approach to Multiple Levels of Hierarchy

The controllers `top` and `bottom` allow for hierarchic contract interpretation with only two levels of hierarchy. To extend the approach to multiple levels, another kind of controller should be used in the intermediate levels. Before describing this type of controller, let us see the global interaction between the controllers. Figure 6.13 describes a hierarchy of contract interpreters. At the head of the tree, there is a controller of type `top`. At the leaves, controllers of type `bottom` manage the explicit contracts. In the intermediate levels, controllers of type `middle` are used.

The idea is that each controller constructs on the fly, the transitions needed by the controller above it. Using the explicit contracts, $\texttt{bottom}_2$ and $\texttt{bottom}_3$ construct the contracts needed by the controller $\texttt{middle}_2$. The controllers $\texttt{middle}_1$ and $\texttt{middle}_2$ construct the contracts needed by `top` based on the transitions received form the controllers they interact with.

```
1  Controller top is{
2  C ⊂ ⋃_i C_i // the set of subcomponents
3  Available ⊂ ⋃ ID_i
4  NTC ⊂ (N × ⋃ T_i × C)
5  for op_G do{
6       forall c in C {c.enq;}
7       let NTC = {(n,t,c) | ∃c ∈ C.(n,t) ∈ c.resp}
8       let TR = {t | ∃n,c.(n,c,t) ∈ NCT}
9       let (n,c,t) ∈ NCT.t = (s,lab,s') ∈ Ex^TR_Available
10      ∀ id ∈ req_t {id.put; id.get;}
11      c.ic_(n);
12      Available := (Available\req_t)∪(prod_t)
13 }}
```

Fig. 6.12: Sketch of controller at the top of the hierarchy

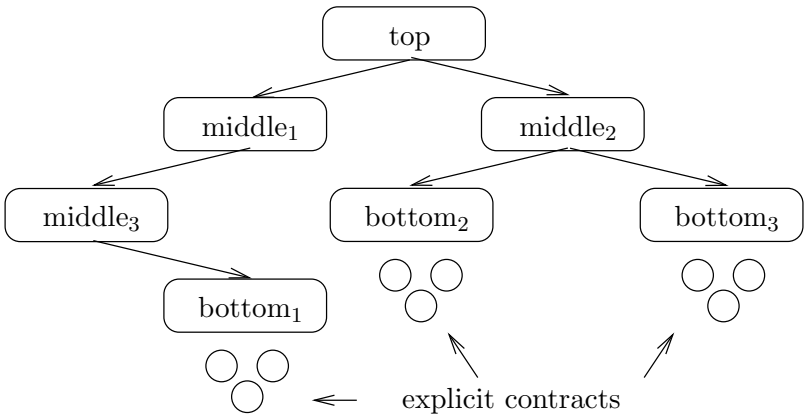


Fig. 6.13: Hierarchic contract interpreters

The transition to execute is selected by the controller `top`. Once selected, `top` activates the corresponding component. The controller of type `middle` inside this component translates it into the consequent activation of its subcomponent. The sequence of activations goes in depth until reaching the activation of a basic-component by a controller of type `bottom`.

Figure 6.14 is the sketch of the controllers of type `middle`. It has some similarities with the controllers `top` and `bottom`. When activated with `enq`, the controller constructs the set NTC: it asks the subcomponents for their possible transitions through *enq* (line 8); the set NTC will contain the set of indexed transitions and their corresponding component. The indexed transitions are gathered from the output control ports *resp* of each component (line 9). At line 10, it computes the set TR_I, it is the set of transitions projected on the local inputs. The set *ret* is the projected transitions on the global ports $Proj^t_G$ and their indexes, we keep only the transitions that have their local dependencies resolved ($Proj^t_I \in Ex^{TR_I}_{Available}$). At line 12, the controller assigns the set *ret* to the control output *resp*.

When activated with $\texttt{op}_{(n)}$, the controller selects the transition and the component associated with n. It provides the component with the required inputs and activates it (lines 16,17). The global data and control outputs are produced depending on the values provided by the components (line 18,19). At the end, the set of available inputs is updated.

```
1  Controller middle is{
2  C ∈ ⋃_i C_i // the set of subcomponents
3  Available ⊂ ⋃ ID_i
4  NTC ⊂ (N × ⋃T_i × ⋃C_i)
5  G = ID_G ∪ OD_G ∪ IC_G ∪ OC_G
6  I = ⋃_i(ID_i ∪ OD_i ∪ IC_i ∪ OC_i)
7  for enq do{
8       forall c in C {c.enq;}
9       NTC = {(n,t,c) | ∃c ∈ C.(n,t) ∈ c.resp}
10      TR_I = {t_I | ∃(n,t,c) ∈ NTC.t_I = Proj_I^t}
11      ret = {(n,t') | (n,t,c) ∈ NTC.t' = Proj_G^t ∧ Proj_I^t ∈ Ex_Available^TR_I}
12      resp := ret;
13 }
14 for op_(n) do{
15      let t = (s,lab,s').∃(n,t,c) ∈ NTC
16      forall id in req_t {id.put; id.get;}
17      c.ic_(n);
18      forall od in (prod_t\I) {id.put; id.get;}
19      forall oc in (ocs_t ∩ G) {oc := c.oc;}
20      Available := (Available\req_t)∪(prod_t)
21 }}
```

Fig. 6.14: Sketch of controller at the middle of the hierarchy

6.2.2.7 Contracts Interpretation and Master/Slave Relation

The master/slave relation introduced in Section 3.4.2.2 may be taken into account when interpreting contracts. This relation implies a notion of atomicity in the activation of components. It relates an output control port of an initiator component to the input control port of the target one. If the control output port is produced when activating the initiator, the target component should be activated with the corresponding control input during the same macro-step.

The relation is defined over a set of components $\{C_i = (\Sigma_i, \Sigma_i^{\text{init}}, ID_i, OD_i, IC_i, OC_i, \mathcal{B}_i)\}_I$. The subset of control inputs and outputs (and their corresponding components) involved in the master/slave relation are defined by the sets $tar \subseteq \bigcup_I C_i \times \bigcup_I IC_i$ and $init \subseteq \bigcup_I C_i \times \bigcup_I OC_i$ respectively. The master/slave relation is defined as a function $arch : init \rightarrow tar$.

6.2.2.8 Example

Figure 6.15 is an example modeling function call and return in 42. The system is composed of a component C that may call the function `f` provided by the component C'. The master/slave relation is described at the top of the component MAIN. It expresses that: whenever the control output $\texttt{call}_f$ of the caller C is produced, the component C' should be activated with `f`, and whenever the callee's output control port $\texttt{end}_f$ is produced, the caller should be activated with $\texttt{cont}_f$.

Components and their Contracts The interface of the caller C exposes one output data port `param` to send function call parameters, and one input data port `return` to receive the results of the call. It also exposes two input control ports `op` and $\texttt{cont}_f$: `op` is used to make it perform one simulation step, $\texttt{cont}_f$ is used to activate it after a function call. The output control port $\texttt{cont}_f$ is used to tell the controller that a function should be called. The contract of C is described in the Figure 6.16 (the one on the left). It expresses that the component should be activated first with `op`; the component then produces the value of the control output $\texttt{call}_f$ and the value of the `param` data port. Second, the component should be activated with $\texttt{cont}_f$

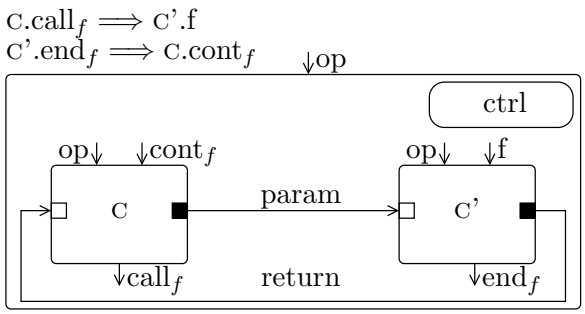


Fig. 6.15: Function call and return modeling in 42

which requires the result of the call to be available in the input data port `return`.

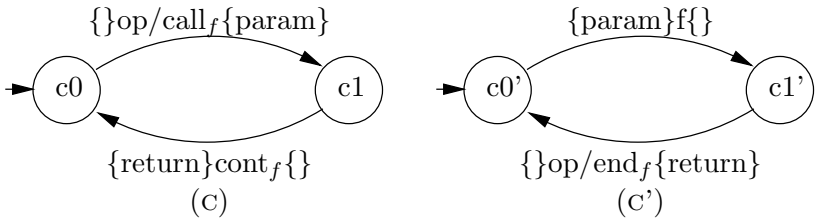


Fig. 6.16: Contracts for the components in Figure 6.15

The interface of the callee C' exposes one input data port `param` to receive function call parameters, and one output data port `return` to provide the results of the function call. The input control port `op` makes it perform one simulation step; `f` is used to perform the function call. The output control port `end`$_f$ is used to tell the controller that the function terminates. The contract of C' is described in the Figure 6.16 (the one on the right). It expresses that the component should be activated with `f`, and requires the input `param`. Then, the component should be activated with `op`, which produces the value of the control output `end`$_f$ and provides the results of the function call through `return` data port.

The Controller The controller interpreter needed for this kind of application is described in Figure 6.17. It knows about the components and their contracts CT. It manages the state of the contracts (CS) and the available inputs ($Available$). It also needs the function describing the master/slave relation $arch : init \rightarrow tar$. The controller is somewhat similar to the one described in Figure 6.9, with the small difference that it manages the master/slave relation.

The set of executable transition TE is also computed differently. It is composed of the set of transitions having their requirements satisfied, and the corresponding control input is not subject to the master/slave relation (line 10).

For each macro-step, at least one component is actviated. The first actviation is at line 13. The actviated component may produce some control outputs which correspond to initiators in the master/slave relation. The produced control outputs are maintaned in the set *calls* (line 16). As long as the set contains some elements, the controller retrieves one of them and selects a transition corresponding to the taraget activation (lines 18 to 22). The controller activates the corresponding component (lines 23 and 24), and updates the set *calls* in case some control outputs are produced (line 25). In each iteration the states of the contracts and the available inputs are updated.

Comments The hierarchic controllers interpreters described in Figures 6.11, 6.12, 6.14 may be extended such that each of them manages a local master/slave relation. To do so, we extend

```
1  Controller interpreter is{
2  CT ⊂ (⋃ C_i × ⋃ T_i) //Components and their transitions
3  Available ⊆ ⋃ ID_i // set of available input data
4  CS ∈ ΠS_i // Contracts states
5  tar ⊆ ⋃ C_i × ⋃_I IC_i
6  init ⊆ ⋃ C_i × ⋃_I OC_i
7  arch : init → tar
8  for op_G do{
9       let TR = {t | t = (s, lab, s') ∈ ⋃ T_i.s ∈ CS}
10      let ET = {t | ∃(c, t) ∈ CT ∧ t ∈ Ex^TR_Available ∧ (c, ic_t) ∉ tar}
11      let (c, t) ∈ CT.t = (s, lab, s') ∈ ET
12      forall id in req_t {id.put; id.get;}
13      c.ic_t;
14      Available := (Available\req_t)∪(prod_t)
15      CS := CS\{s} ∩ {s'}
16      let calls = {(c, oc_1), (c, oc_2), ... | oc_i ∈ ocs_t ∩ init}
17      while (calls ≠ ∅) do{
18        let (c, oc) ∈ calls
19        calls = calls/(c, oc);
20        let TR = {t | t = (s, lab, s') ∈ ⋃ T_i.s ∈ CS}
21        let ET = {t' = (ss, lab', ss') | t ∈ Ex^TR_Available ∧ ((c, oc), (c', ic'_t)) ∈ arch}
22        let (c', t') ∈ CT.t' = (s, lab, s') ∈ ET
23        forall id in req'_t {id.put; id.get;}
24        c'.ic'_t;
25        calls := calls ∪ {(c', oc_1), (c', oc_2), ... | oc_i ∈ ocs'_t ∩ init}
26        Available := (Available\req'_t)∪(prod'_t)
27        CS := CS\{s} ∩ {s'}
28      }
29 }}
```

Fig. 6.17: Managing master/slave relation when interpreting contracts

```
1  Controller interpreter is{
2  CT ⊂ (⋃C_i × ⋃T_i) //Components and their transitions
3  Available ⊂ ⋃ID_i // set of available input data
4  CS ∈ ΠS_i // Contracts states
5  for op_G do{
6       let TR = {t | t = (s, lab, s') ∈ ⋃T_i.s ∈ CS}
7       let ET = Ex^TR_Available
8       if (ET = ∅) {deadlock}
9       ...
10      ...
11 }}
```

Fig. 6.18: Detecting deadlocks when interpreting contracts

their code with the loop described between lines 15 and 24 of the controller code in Figure 6.17, together with the small difference that holds when computing the set of executable transitions.

6.2.2.9 Runtime Verification of Properties

Contract-based simulation of asynchronous systems may serve for runtime verification of properties (see Section 2.4.3). The organization of a simulation model in 42 makes it easy to decide how a property should be encoded.

For instance, if the property is related to some data communicated between components, the best decision is to encode the property as a component. This component is then connected to some outputs of the other components and activated by the controller.

On the other side, if the property is related to control, the best decision is to encode it into the controller. For instance, in order to check component synchronization or the absence of deadlocks.

Detecting Deadlocks Within the 42 simulation engine, a *deadlock* corresponds to the state where no further simulation step can be performed, i.e., no transition is executable by the controller. One may check for *deadlocks* simply by observing the set of the executable transitions, computed by the controller at each simulation step.

Figure 6.18 illustrates part of the controller interpreter described in Figure 6.9 augmented with statements to check potential deadlocks. We recall that the state ET is the set of executable transitions. If the set ET is empty, no transition can be selected by the controller, thus no further simulation steps. This corresponds to a *deadlock*. The statement at lines 8 is added to the controller code in order to detect such a situation.

Chapter 7

Hardware Simulation and Software Execution

Introduction (En) *A modeling tool for embedded systems should encompass the modeling of the hardware and the software parts. Besides, it should enable the interaction between the two parts. This chapter deals with the development of virtual prototypes of the hardware to enable the execution of the embedded software. At a first place, we introduce 42 as a model for describing hardware prototypes, and the way the embedded software may be run on the virtual prototypes. At a second place, we present first steps toward the formalization of SystemC/TLM components with 42. SystemC/TLM is the de-facto standard for the modeling of* Systems-on-a-Chip *but lacks in formalization.*

Contents

Introduction (Fr) *Un outil de modélisation pour les systèmes embarqués doit permettre la modélisation de la partie matérielle, de la partie logicielle, ainsi que l'interaction entre les deux parties. Dans ce chapitre, nous montrons des cas d'utilisation de 42 dans le domaine du prototypage virtuel du matériel pour l'exécution du logiciel embarqué. Dans un premier temps, nous donnons un exemple de prototypage virtuel de systèmes matériels et comment exécuter le logiciel sur ces prototypes 42, indépendamment de tout autre approche existante. Ensuite, nous nous intéresserons à la 42-isation de SystemC/TLM ; SystemC/TLM étant un des standards de développement de prototypes virtuels pour les systèmes-sur-puce. La 42-isation consiste à décrire les composants SystemC/TLM en 42. Notre approche contribue à la formalisation des composants TLM.*

7.1 Prototyping Hardware Using 42

7.1.1 An Example *System-on-a-Chip*

In order to start writing the software early in the design flow of a *System-on-a-Chip*, software developers require some executable prototype of the hardware platform. As a virtual prototype, one may use the RTL model of the hardware platform. But, for the sake of efficiency, the virtual prototype should be abstract so as to expose only the behaviors of the hardware that are interesting for the software developers (see Section 2.2.1). The virtual prototype may be useful to detect some design errors related to the hardware platform earlier, or to perform architecture exploration.

This section is presented as an example to expose some guidelines to model virtual prototypes of the hardware with 42.

7.1.1.1 A Typical Hardware Architecture for *Systems-on-a-Chip*

Figure 7.1 is the structure of a hardware platform. It is made of a CPU, a LCD (Liquid Crystal Display), a bus, and a memory. The LCD is a component that may be programmed by the CPU, in order to perform repetitive transfers from the memory (like a DMA, Direct Memory Access component). There is a need for some communication between the LCD and the CPU, to inform the CPU that the transfers that were programmed are finished. This is done with an interrupt. In order to display something on the LCD, the software writes an image in some dedicated place of the memory; then it programs the LCD so that it now transfers the image from the memory to the screen; then it waits for the interrupt from the LCD, meaning the transfer is finished. Figure 7.3 describes part of the software that will run on the CPU. We give more details about the software in Section 7.2.

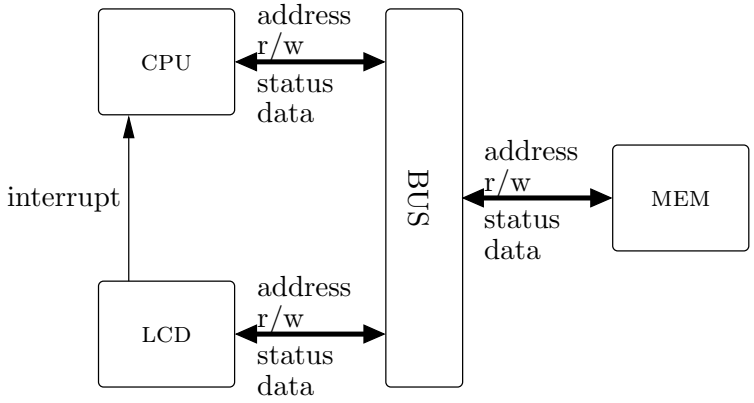
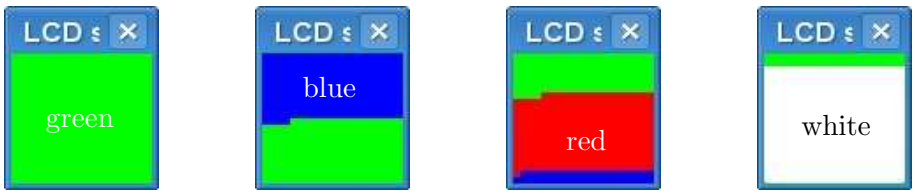


Fig. 7.1: An example of hardware architecture for *Systems-on-a-Chip*

7.1.1.2 Intended Behavior

In our example, the software repeatedly displays a full green screen, then a blue one, then a red one, and so on. On Figure 7.2, (a) and (b) show normal states of the display. (b) is possible because the LCD may take some time to replace a full green screen by a full blue one. (b) corresponds to the following situation: the CPU had written a green image before, it has just written a blue one; it is waiting for the interrupt telling it that the blue transfer is finished. The LCD is transferring the blue image, part of the green one is still visible. When the transfer is finished, and the screen is totally blue, it will send the interrupt to the CPU.

On the contrary, (c) should not be possible. The only way of obtaining such a state is when the



a b c d

Fig. 7.2: LCD ScreenShots

```
...
int main(int argc, char **argv) {
    while(1) {
      lcd_print(green);
      write_lcd(0x01,0x1);
      wait_interrupt();
      lcd_print(blue);
      write_lcd(0x01,0x1);
      wait_interrupt();
      lcd_print(red);
      write_lcd(0x01,0x1);
      wait_interrupt();
    }
}
```

Fig. 7.3: Part of the software running on the CPU

software starts reprogramming the LCD (writing the red image, for instance) without waiting for the last programming to finish. The situation (d) is due to some data errors of the software.

7.1.1.3 Typical Bugs

A typical misconception of the hardware platform is to forget the interrupt wire between the LCD and the CPU. If this wire does not exist, the only way for the software to know that a transfer is finished would be to have a precise knowledge of the *time* it can take. In high level models like TL models, this is prohibited. Synchronizations should be made explicit, so as to get robust software, able to run correctly on various hardware platforms, with different timings. A typical synchronization bug in the software is to forget to wait for the interrupt from the LCD. Another kind of bug would be due to pure *data* errors, like writing to an erroneous part of the memory, or writing only a part of the image, or using the wrong color, etc.

7.1.1.4 Benefits of Using 42 for Modeling Hardware

The benefits of a 42 model for the system described above are the following:

Contract-based simulation to detect synchronization problems: the whole system can be described by its architecture and the contracts of the components, without knowing the details of the components. Then, the system can be simulated following the principles of Section 6.2.2. In this first step, the contract of the CPU is in fact the contract of the CPU plus the software that will run on it; but the part of the behavior which is due to the software is very abstract, as we will see on the example.

Executing the real software on the simulated hardware platform: when the architecture and the contracts have been simulated so that early synchronization problems (like

forgetting the interrupt wire, or forgetting to wait for the interrupt in the CPU+SW contract) have been discovered and corrected, the same model can be simulated together with the execution of the real software. This allows us to see more bugs, typically the data bugs (wrong color, etc.). But the interesting part is that it allows a check on the compatibility between the contract of the CPU component (which includes some information on the software) and the actual piece of software, written in C or other languages.

7.1.2 Modeling the Hardware Architecture with 42

Figure 7.4 is the structure of the model in 42. Each wire between two components models a communication in the real hardware platform. The type of the wires may be Boolean (e.g., for `intr`) or record (e.g., $\texttt{acd}_X$ encapsulating the address, the control R/W, and the data to be written). Each component is equipped with its local contract where the output control values may be used. LCD`.report` may take the value `ok` (resp., `ko`) which states that the transfer of the image from the memory is finished (resp., not finished).

Ports Type:
CPU.report : $\{MT, LT\}$ **acd$_X$** : $[a : int, c : \{R, W\}, d : int]$ **intr** : $\{t, f\}$
CPU.report2 : $\{IT, NoIT\}$ **resp$_X$** : $[status : bool, data : int]$ **target** : $\{L, M\}$
LCD.report : $\{ok, ko\}$

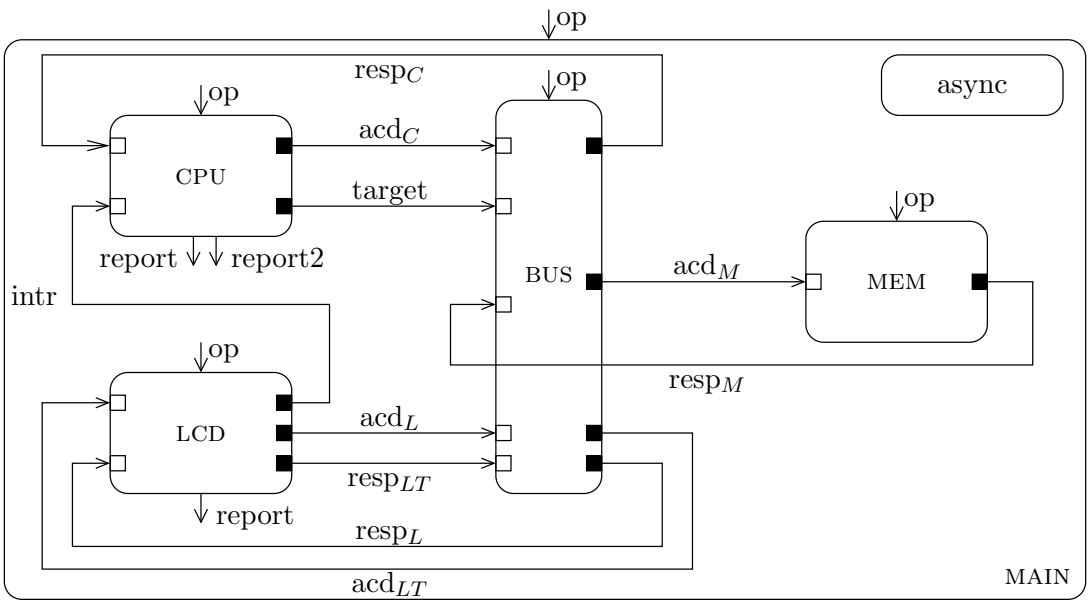


Fig. 7.4: The 42 model of the example in Figure 7.1

7.1.2.1 Describing Components with Control Contracts

The Contract of the CPU is illustrated in the Figure 7.5. It is in fact the contract of the embedded software, plus some hardware mechanisms like the memorization of the interrupts until they are taken into account (variable `interrupt` in Figure 7.10). The contract of the CPU reports on the state of the software, which may end an atomic step in three cases: either it stops just before an access to the `memory` or the `LCD`, or it is waiting for an interrupt. These three situations are encoded with the two control outputs `report` and `report2`. The control output `report` may take its value in { `MT, LT` } (memory or LCD access, respectively). The control output `report2` may take its value in { `IT, NoIT` }. The value `IT` indicates that the software is waiting for an interrupt.

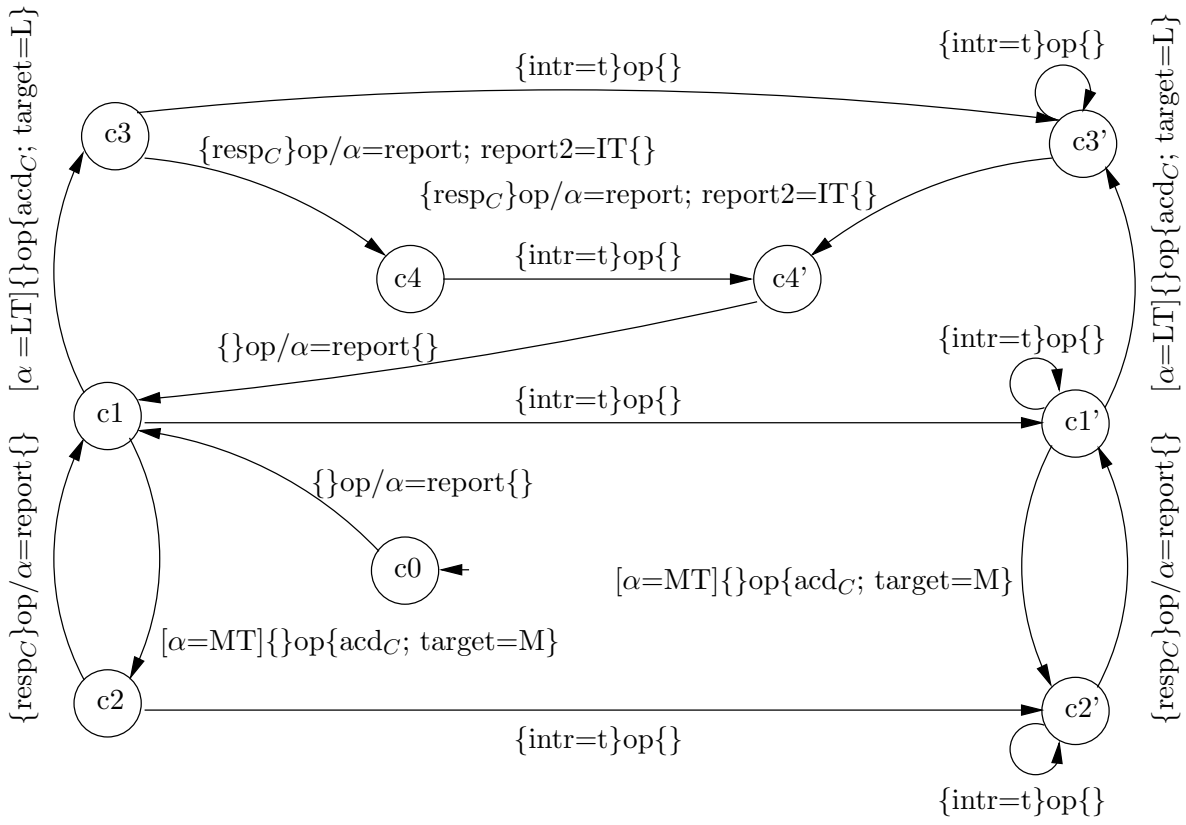


Fig. 7.5: Contract of the CPU component (software + interrupt bit)

The contract of the CPU is as follows. The unprimed states (resp., the primed ones) correspond to cases when there is no memorized interrupt (resp., there is a memorized interrupt). When the interrupt arrives (transitions labeled by {intr=t}op{}), the contract changes from an unprimed state to a primed one (e.g., c4 to c4'). When the interrupt is taken into account, the contract changes from a primed state to an unprimed one (the only one is from c4' to c1, the interrupt bit may not be cleared before receiving the acknowledgment corresponding to the last LCD programming).

From the initial state c0, the initial activation op goes to state c1 and corresponds to the first part of the software, before it stops for a memory or LCD access (at this point, it *should not* stop because it is waiting for an interrupt). The value of report output referred to by α indicates the target (MT or LT).

From c1, the interrupt can be taken, and the contract goes to c1'. Otherwise, the software starts the memory or LCD access, by sending relevant information on its output data ports ($\texttt{acd}_C$, target). It goes to c2 or c3.

In c2 and c3, the CPU is waiting for the acknowledgment from the target component ($\texttt{resp}_C$). From c2 it goes back to c1 for potential new memory accesses; from c3 it goes to c4: the LCD has been programmed, and the software should now wait for the interrupt stating that the LCD has finished.

From c4, the only possible change is that the interrupt arrives, and is stored. The contract goes to state c4'. Then the software can take it into account, and go to state c1 again.

At states c3 and c3', the software is waiting for the acknowledgement from the LCD. Once the acknowledgement is received, the transition of the contract states that the software is waiting for an interrupt by explicitly setting the control output report2 to the value IT.

The Contract of the LCD is illustrated in the Figure 7.6, it describes the following behavior: in state l0 the LCD waits until it is programmed by the CPU (this comes as a data on input $\texttt{acd}_{LT}$, transition to l1). Then it acknowledges this by writing to its port $\texttt{resp}_{LT}$ and it reaches state l2. The loop between l2 and l3 corresponds to a sequence of read actions from the memory ($\texttt{acd}_L$), each of them being acknowledged ($\texttt{resp}_L$). For each read the contract stores the control output report in variable β. In state l3, if β is ok it means this was the last read, and the LCD returns to state l0 and writes a true value to its interrupt port (intr=t), for the CPU. If β is ko, it writes a false value (intr=f) and continues.

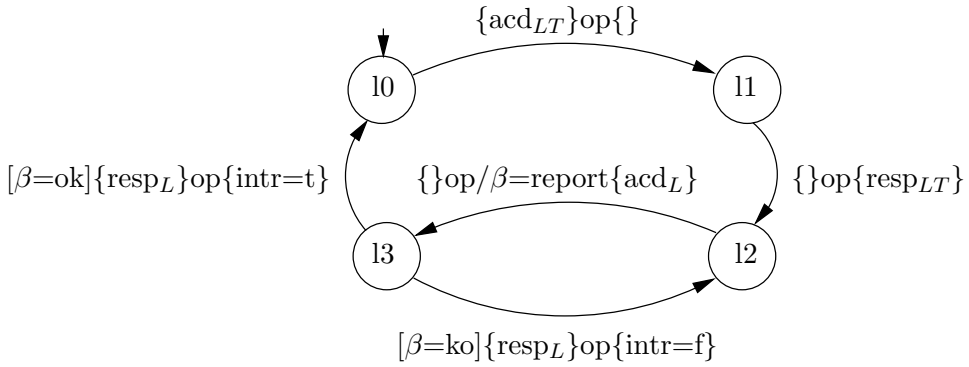


Fig. 7.6: The contract of the LCD

The Contract of the Component MEM is illustrated in the Figure 7.7. It is quite simple: it accepts read or write requests on its input port $\texttt{acd}_M$ and acknowledges them by $\texttt{resp}_M$ (which encapsulates the request status, and potentially a data delivered for a read request).

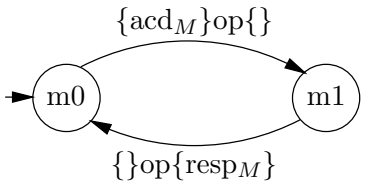


Fig. 7.7: The contract of the memory

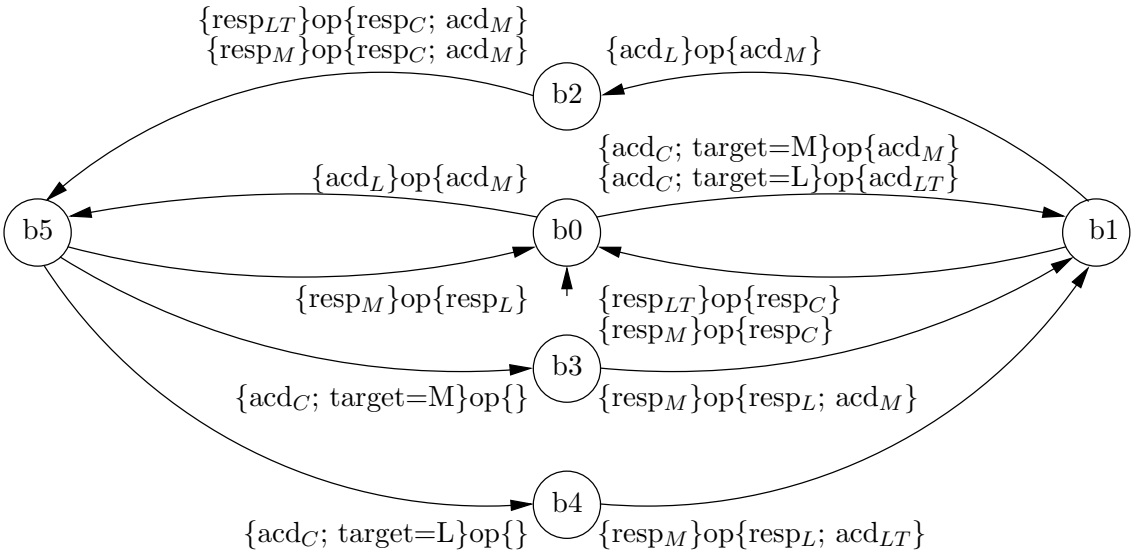


Fig. 7.8: The contract of the BUS

The Contract of the Component BUS is illustrated in the Figure 7.8. It is more complex, because it describes the correct transportation of read and write accesses, and their corresponding acknowledgments. The possible transfers are: the CPU writes to the memory; the LCD reads from the memory. We do not consider the case of parallel software running on the CPU and issuing several accesses to the bus in parallel. The contract of the BUS shows that an access to the bus can be memorized until the current transfer is terminated, but only two parallel transfers are considered. In a real bus, this is far more complex, but the contract is of the same form (we would need a better language than flat explicit automata to describe it, though).

7.1.3 Contract-Based Simulation

Following the principles of contract interpretation described in Section 6.2.2, the hardware platform can be simulated with the contracts alone. The contracts are considered as an abstract implementation of the hardware components. But for the contract of the CPU we include a very abstract view of the embedded software.

At this point we simulate only the contracts in order to check component synchronization; several bugs are due to synchronization problems. Executing contracts allows for discovering them early. Such bugs would be hard to detect when dealing with several lines of implementation code. The bugs may be related to the software abstract view (included in the contracts of the CPU) or even related to the design of components modeling hardware components.

An exhaustive simulation would build the complete interleaving graph (for applying model-checking), but it can also be used as an input for a runtime verification tool (in the sense of a tool like Verisoft [God97], or using dynamic partial orders adapted to SystemC/TLM [HMMCM06]). Even if we do not use a specification language for temporal properties, we can observe *generic*

properties like deadlocks (see Section 6.2.2.9), and some livelocks. The two following simulation results, obtained for the case study, illustrate two bugs that can be found early.

The Interrupt Bug After writing an image, the processor waits for an interrupt coming from the LCD to start writing a new image. If the interrupt never occurs, the system is blocked. Suppose we modify the contract of the LCD to introduce this bug: on Figure 7.6, the transition from state l3 to l0 is now labeled by: `[`β`=ok]{`$\texttt{resp}_L$`}op{}`. Suppose the simulation has reached the state {`(c4,b0,m0,l3)`, *Available* =`{`$\texttt{resp}_L$`}`, β`=ok`} [1] (the processor is waiting for the interrupt, the memory is waiting for a READ/WRITE request, and the bus has just delivered the memory acknowledgment to the LCD). At this state, only the LCD may be activated, and the simulation moves to the state {`(c4,b0,m0,l0)`, *Available* =`{}`}. At this state, all the component transitions require inputs. But no inputs are available, which leads to a deadlock.

Other Bugs Other problems may be detected by the simulator. For instance, there are cases when a component is never activated. It's not necessarily a bug, but it deserves at least a warning. For example, when the LCD waits to be programmed, if the processor never does it, the LCD is never activated.

7.2 Software Execution

Figure 7.9 illustrates a possible code for the software that will run on the processor of the hardware platform described earlier. The software describes a periodic behavior: writing an image on some place in the memory, pixel by pixel using the function `write_mem()`; then programming the LCD using the function `write_lcd()` so that the LCD starts transferring the image; then waiting for the interrupt coming from the LCD by calling the function `wait_interrupt()`, in order to start writing the next image.

On the real system, the functions like `write_mem()` and `write_lcd()` will be implemented (later) by accesses to the memory and the LCD registers via the bus in order to write something. The software developers have to use such functions because at this level of the design process, the details of the hardware platform may not be known. Here we focus on the global behavior of the software, not on the details of the communication with the hardware components.

7.2.1 Using Wrappers for Hardware/Software Simulation

To reflect the decisions of the software on the hardware platform, one has to execute the embedded software on the simulated hardware. This may be done in two ways: using a ISS or a native wrapper. Both of the two techniques were discussed in Section 2.2.2.2. In this section, we propose to make the software implementation execute in a *native wrapper*.

7.2.1.1 Wrapping the Software into a 42 Component

The wrapper provides the implementation of the functions used by the software to communicate with the hardware components; i.e., `write_mem()`, `write_lcd()`, and `wait_interrupt()`. The code of such a wrapper is given in Figure 7.10.

To make the software run together with the simulation of the hardware platform, the wrapper is implemented as a 42 component. To each function used by the software, the wrapper associates

[1] The set *Available* contains the available inputs. See Section 6.2.2 for more details.

```
#define WIDTH   20
#define HEIGHT  20
#define blue    0xff0000ff
#define red     0xffff0000
#define green   0xff00ff00

void lcd_print (unsigned long int pattern ) {
  int y;
  for (y=0; y<HEIGHT*WIDTH; y++)
     write_mem(y, pattern);
}

int main(int argc, char **argv) {
    while(1) {
      lcd_print(green);
      write_lcd(0x01,0x1);
      wait_interrupt();
      lcd_print(blue);
      write_lcd(0x01,0x1);
      wait_interrupt();
      lcd_print(red);
      write_lcd(0x01,0x1);
      wait_interrupt();
   }
}
```

Fig. 7.9: A possible software code for controlling the LCD

```
interrupt : bool;
write_lcd(int a, int d){
  report.write("LT");
  report2.write("NoIT");
  pause();
  acd_c.write(a,"W",d);
  target.write("L");
  pause(); resp_c.read();
}

wait_interrupt(){
  report2.write("IT");
  pause();
  interrupt=0;// clear interrupt
}

write_mem(int a, int d){
  report.write("MT");
  report2.write("NoIT");
  pause();
  acd_c.write(a,"W",d);
  target.write("M");
  pause(); resp_c.read();
}

op(){
 if(intr.hasValue())
  intr.read();
  interrupt=1;// set interrupt
 else  resume();
}
```

Fig. 7.10: The code of the wrapper component

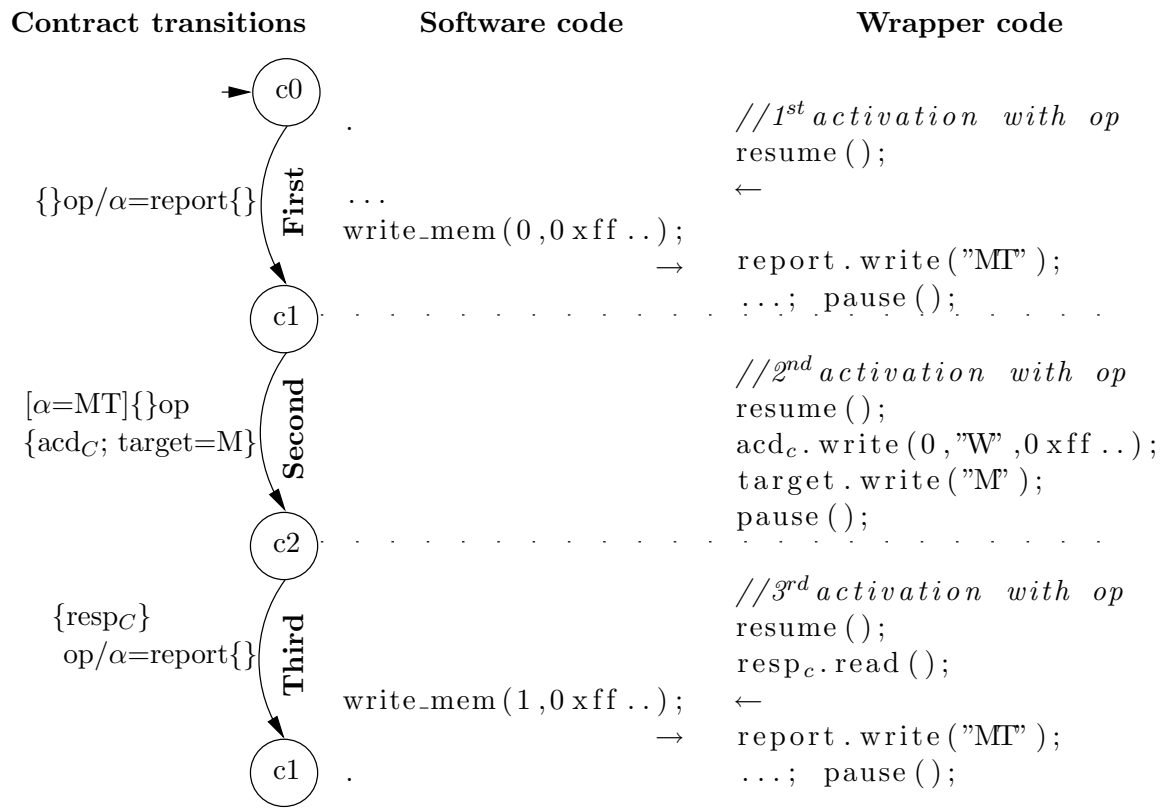


Fig. 7.11: The correspondence between the software code and the contract transitions

a set of actions to accomplish the decision of the software. The combination of the wrapper plus the embedded software defines the implementation of the processor component (CPU).

The control input `op` from which the CPU is activated is defined by the wrapper. When activated with `op`, the wrapper lets the software execute a piece of its code. When the software calls the wrapper functions, the wrapper states on the decision of the software putting a value on `report` and `report2` output control ports.

For instance, when the software performs the call `write_mem()`, the wrapper reports through the control outputs that the software is indeed willing to write something on the memory. It also puts the parameters of the function call on the corresponding output data ports; i.e., the memory address, the data to be written, etc.

7.2.1.2 Activating the Wrapper to Make the Software Execute

Technically, the wrapped software is run as a thread that may suspend itself with `pause()`. The simulation of the hardware platform is the main program; it may reactivate the software thread with the `resume()` function.

Consider a case when the simulation controller transforms a global activation `op` into an activation of the processor component (`CPU.op`). This resumes the thread of the software, which runs until the next `pause()` in a communication primitive (e.g., `write_mem`). The software thread being suspended, the simulation controller considers `CPU.op` to be terminated. Next time it will be activated, the software thread will start execution from where it had paused. An execution from one pause to the next one is the *atomic* step of the CPU component.

7.2.1.3 An Example of Execution Trace

Figure 7.11 illustrates an execution trace of the wrapped software starting from the instantiation of the component CPU. On the left of the figure, we describe the transitions of the contract of the CPU; on the right, the code executed with respect to each transition.

First, the thread executing the software is paused after the instantiation. The activation of the CPU (the wrapper) will resume the thread. The control passes to the software which executes some code until calling the function `write_mem(0,0xff..)` to write a value at the address `0`. The wrapper reports that the software requires memory access through the control output `report` (outputting `MT`). After that, the thread is suspended and the control is handed back to the controller that may activate other components.

Second, for the 2^{nd} activation of the CPU, the thread is resumed. We are still in the body of the function `write_mem()`. As described by the transition of the contract, the wrapper outputs the memory address, the value to be written, and the target component (i.e., `M`). The output values are those corresponding to the parameters of the function call. At the end, `pause` is called.

Third, once the response of the memory access is made available for the CPU, the controller activates it. This resumes the execution of the thread. The wrapper reads the response. At this point, the call to the function `write_mem(0,0xff..)` is terminated, the control flow is back to the software implementation. The software performs the call `write_mem(1,0xff..)` to write the next pixel into the memory. The wrapper behaves as in the first activation.

7.2.2 Checking Software Implementation

Once the bugs related to component synchronization are detected by the simulation of the contracts alone, the execution of the actual software together with the contracts of the components may reveal other bugs. First, the behavior of the software may not conform to what is described by the contract. Checking the software consistency with the contract allows for detecting such implementation errors. Second, observing the simulation of the HW/SW components may also reveals other bugs.

7.2.2.1 Using the Contracts to Check Software Decisions

When executing the contracts alone, the values of the control outputs are non-deterministic, except for those associated with explicit values. When running actual embedded software, we check its consistency with the contract of the CPU as described in 6.1.1. The production of the control outputs is done by the wrapper functions. Hence checking the control output of the wrapper component is a way of checking that the software has indeed made a call to the function it was expected to call.

For instance, one of the typical bugs mentioned previously is: the software omits to wait for the interrupt before reprogramming the LCD. So, suppose we omit the first occurrence of `wait_interrupt()` in the program of Figure 7.9. The simulation will report that the contract is violated. The outgoing transitions from `c3` and `c3'` declare that the control output `report2` will take the value `IT`. However, the only possible way to output the value `IT` on the control output `report2` is to call the function `wait_interrupt()` which is omitted by the software. This is a bug in the software, detected because it is not consistent with its contract.

This mechanism makes it possible to run the software and to check its consistency with the contract, even if it is given as object code. The only constraint is that it uses the wrapper

functions to access the hardware.

7.2.2.2 Observing the HW/SW Simulation to Detect More Bugs

Despite the fact that the software respects its contract, there may be more bugs, related to the data. For instance, suppose the code of the `lcd_print` function is changed in the program of Figure 7.9: the condition of the loop is now `y<HEIGHT+WIDTH`, which means that the program writes 40 pixels instead of 400. This bug may be detected by observing the output on a simulated LCD, as shown on Figure 7.2-(d). The top of the LCD is colored whereas the bottom of it has the color corresponding to the initial memory value (white).

7.3 Formalizing SystemC-TLM with 42 Components

The design of virtual prototypes for *Systems-on-a-Chip* has seen the emergence of the Transaction Level Modeling (see Section 2.2.1.1). The TLM approach is component-based, in which hardware blocks are described by means of *modules* communicating with *transactions*. The de-facto standard for TLM is SystemC which gained a lot of success due to its C/C++ part.

Being based on a general-purpose programming language, SystemC-TLM has no formal semantics. Models written with SystemC-TLM are hard to analyze formally. On the other hand, 42 allows for describing components formally. As introduced in Section 3.4.1, it may be used for reasoning on components of other frameworks.

The purpose of this section is to establish the structural correspondence between the SystemC-TLM components, as defined informally by the TLM guidelines, and the 42 components. The 42-ization of SystemC-TLM components is a way of formalizing the principles of SystemC-TLM. The interesting uses of this approach are explained in section 7.3.3.

To establish the correspondence we use a simple example written in SystemC. The example (denoted by SCTLM in the sequel) is that of Figures 7.12 and 7.13. The corresponding 42 model (denoted by 42M in the sequel) is described in Section 7.3.1.

7.3.0.3 A TLM-PV Example in SystemC

TLM allows several abstraction levels, the example of this section is described at the level of *programmer's view (PV)*. The example models two modules `module1` and `module2` communicating with transactions through the ports attached to each module. The direction of the arrow shown on a port determines the role of the port. On Figure 7.12, `p2` in `module1`, and `p1` in `module2`, are *initiator* ports; `p1` in `module1`, and `p2` in `module2`, are *target* ports.

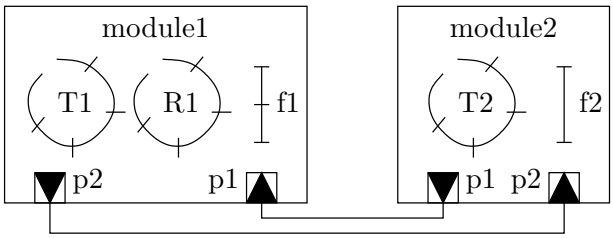


Fig. 7.12: Example of SystemC/TLM (architecture)

Modules implement behaviors The behavior of a module is given by a set of *threads* (represented by circles with *steps* on Figure 7.12), and a set of *functions* (represented by straight

```
1  void module1::T1(){
2    int a = 0;
3    while(true){
4      wait(e1);
5      a++;e2.notify();
6      a++;e3.notify();
7   }}
8  void module1::R1(){
9    int b = 0;
10   while(true){
11     b++;wait(e2);
12     b++;p2.f2(b);
13  }}
14 void module1::f1(int x){
15   cout << x ;
16   e1.notify();
17   wait(e3);
18 }// ** module1 **
```

```
19 void module2::T2(){
20   int c;
21   while(true){
22    c++;  p1.f1(c);
23    c++;  wait(e4);
24   }
25  }
26
27 void module2::f2(int x){
28    cout<< x ;
29    e4.notify();
30 }// ** module2 **
```

Fig. 7.13: Example of SystemC/TLM (code)

lines with steps), both programmed in full C++ (see Figure 7.13). `module1` has two threads `T1` and `R1`, and one function `f1`, while `module2` has a single thread `T2` and a function `f2`.

The threads are *active* code, to be scheduled by the global scheduler; The functions `f1` and `f2` are passive code, waiting to be called. They are attached to the port of the modules. `f1` (resp., `f2`) is attached to the port `p1` (resp., `p2`) in `module1` (resp., `module2`).

Threads and functions of the same module share events Events are used in order to synchronize with each other. They can be notified, or waited for. In `module1` (Figure 7.13) there are three events `e1, e2, e3`.

The threads are managed by the SystemC scheduler The SystemC scheduler is non-preemptive: a running thread has to yield, by performing a wait on an event (or on *time*, see comments below, but we will use untimed models). For instance, for the thread `T1` of `module1`, the only point where the thread yields is `wait(e1)`. The execution of `a++; e2.notify(); a++; e3.notify();` is therefore atomic.

Modules communicate with transactions only According to the TLM-PV guidelines, communications between modules cannot use the event mechanism, because this would be meaningless w.r.t. the physical parallelism to be modeled. The only possible communications are called *transactions*, implemented by blocking function calls; the link between a caller and the callee is established through the architecture.

On Figure 7.13, the thread `T2` of `module2` initiates a transaction on its port `p1` (written `p1.f1(c)`). This is a call to the function `f1` in `module1` (which is attached to the target port `p1` of `module1`), because the initiator port `p1` of `module2` is connected to the target port `p1` of `module1`. When the call is executed (`T2` being running), the control flow is transferred to `module1`, until `f1` terminates; then the control flow returns to `module2`, and the execution continues until the next yielding point (`wait(e4)` in the example). Since function `f1` in `module1` waits for the event `e3`, it yields if `e3` is not present; this means that thread `T2` in `module2` may yield because of a `wait` statement in the function it has called in another module. An example

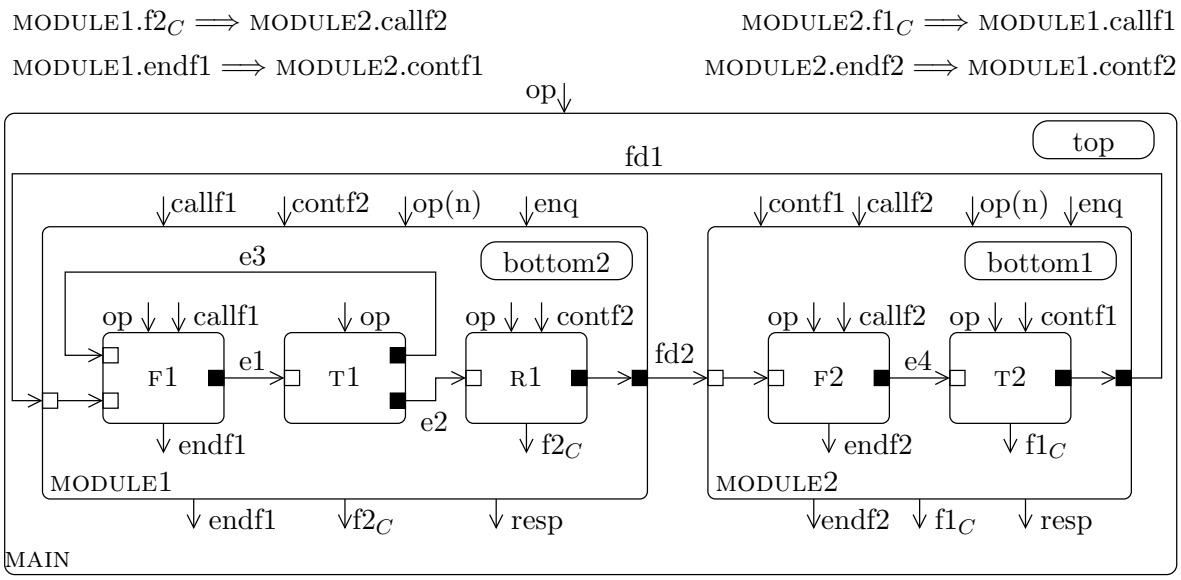


Fig. 7.14: 42 architecture for the system of Figures. 7.12 and 7.13

atomic sequence is the following: the scheduler elects thread `R1`, which is at line 12; it executes `b++` and then calls `f2` via the port `p2`; the body of `f2` in `module2` is executed entirely, and the control returns to `module1`; the thread `R1` loops, executes `b++` at line 11, and stops on `wait(e2)` at line 11.

7.3.1 Structural Correspondence Between 42 and SystemC

In this Section we give the corresponding 42 model (42M) of the SystemC example (SCTLM) of Figures 7.12 and 7.13. The architecture of 42M is given by Figure 7.14; the control contracts associated with the components are described in Section 7.3.2. For the sake of clarity, we will suppose that we cannot have, simultaneously, two active calls of a given function. Our full encoding of SystemC into 42 forbids recursion and relies on code duplication when a function may have several parallel callers.

7.3.1.1 Architecture and Ports

The SystemC model SCTLM is made of several *modules* that have *ports*; each module contains threads and functions. The 42 model 42M is hierarchic; at each level of the hierarchy, it is made of a set of components, and a controller. The architecture of 42M (Figure 7.14) is built by using one 42 component per module in SCTLM, at the highest level of hierarchy; moreover each 42 component corresponding to a module M is itself built as a 42 system, with one component per thread of M, and one component per function of M; The main component only has a control input `op` for asynchronous simulation.

Highest Level The module ports in SCTLM are used to route function calls. At the highest level of the 42 hierarchy, these function calls of SCTLM are encoded by separating the *control* effect, and the *data* exchanged. For instance, the fact that thread `T2` in `module2` may call `f1` of `module1` via the ports `p1` (Figure 7.13), corresponds to:

- The data wire `fd1` from MODULE2 to MODULE1 on Figure 7.14;
- The output control port `f1`$_C$ (for F1 *is called*) of MODULE2;
- The input control port `callf1` of MODULE1 (to activate the code of F1);

- The output control port `endf`$_1$ (for F1 *has finished*) of MODULE1;
- The input control port `contf1` of MODULE2 (to continue the execution of the thread that has called F1, when F1 is finished).
- The master/slave relation to describe how the control should be passed from one module to another (the implications over the assembly).

Each module has output data ports corresponding to the parameters of the functions that can be called from inside the module, and input data ports corresponding to the parameters of the functions it provides (we assume a *union* type for these ports, representing all the functions parameters). It also has one input control `callf`$_i$ for each function it provides, and one control input `contf`$_i$ for each function that may be called from it. Each module also has a control input `enq` (for *enquire*) and a control output `resp` (for *response*). These control ports are used for introspection (see Section 3.2.3). Finally each module has a *parameterized* control input `op` for asynchronous simulation; the integer parameter will be used, in conjunction with the `enq`/`resp` mechanism, in order for `top` to choose a transition in a module.

Deepest Level At the deepest level, the 42 components represent threads and functions that may communicate through events. The events are encoded as data wires[2]. A thread or function that may notify (resp., wait for) the event `e` has an output (resp., input) data port `e`. The controllers `bottom1` and `bottom2` ensure the connection between the control ports of the modules, and the control ports of the functions and threads.

Each component associated with a function has an input data port for the parameters (connected to the corresponding input data port of the module); it also has a control input `callf`$_i$ (used to start the function), and a control output `endf`$_i$ (produced when the function terminates); finally the control input `op` is used to re-activate the function code, when it had stopped on a `wait`, and still has something to do before it terminates. If the function calls another function, the component also has the ports related to functions, explained below for threads.

Each component associated with a thread has a control input `op` to perform one execution step, from where it had stopped last time it was activated, until the next `wait` or function call. For each function `f` the thread may call, the component has an output control port `f`$_C$ to signal that the function is called, and an output data port to send the actual parameters; this data port is connected to the output data port of the module. A thread that can call a function `f` also has a `contf` control input, to resume its execution when the function terminates.

7.3.2 Executable Contracts For SystemC-TLM Components

Figure 7.15 gives example contracts for the threads and functions T1, R1, T2, F1, F2. The contract of a function F always starts with a transition triggered by `callf`, and needs the data input corresponding to the parameters; it is always a loop (to allow for successive executions of the function). If the body of the function does not wait, nor call other functions, the contract is simply a loop on the initial state, producing `endf`. In the example, this is the case for F2. If the body of the function has `wait` statements, the contract has additional states, with outgoing `op` transitions, which need the data input corresponding to the event which is waited for (this is the case for F1).

The contract of a thread is made of transitions triggered by `op` or the `contf`$_i$ corresponding to the functions `f`$_i$ it may call. Each transition that produces `f`$_{iC}$ also provides the data output

[2]This produces the effect of *persistent* events, in the sense that an event is memorized until consumed by another thread executing a `wait`; in SystemC, standard events are *not* persistent: they are lost if no eligible process waits for them. We could encode non-persistent events in 42 as well.

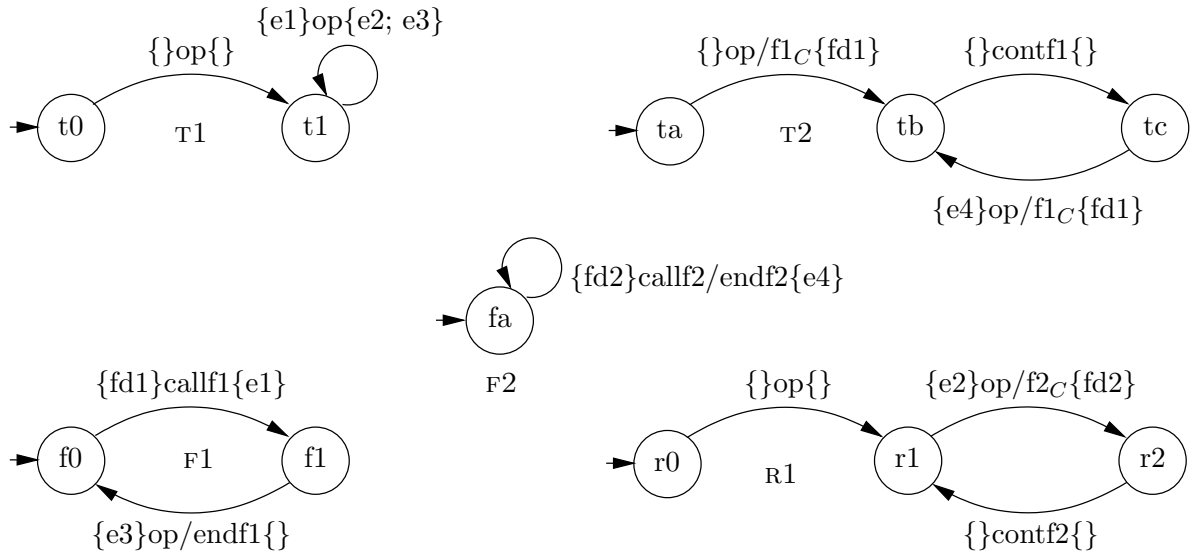


Fig. 7.15: Example 42 contracts for the components of Figure 7.14

corresponding to the parameters, and reaches a state whose only outgoing transition is triggered by `contf`$_i$.

7.3.2.1 Using Hierarchic Contracts Interpreter to Simulate the Model

In the SCTLM model, the threads are managed by the SystemC scheduler, the transactions are simply function calls. In the 42M model, the controllers mimic the behavior of such a scheduler. The *MoCC* implemented by the controllers is *asynchronous*. For modeling SystemC, we made the choice of using controllers acting as contracts interpreters, as those described in Section 6.2.2.4.

The Controller `top` The controller needed at the highest level should be able to manage the asynchronous simulation of components, when the components have to be asked for their specification (i.e., introspection). Also, it should model the general effect of function call and return, taking into account the information provided by the master/slave relation. In Section 6.2.2.7, we gave the implementation details of such a controller.

For a global activation with `op`, the controller `top` asks the modules for the possible transitions in the current states of their contracts (with the control input `enq`). Each module answers through `resp`. Depending on the response of the components the controller translates the global `op` into a sequence of activations of the modules; the first activation corresponds to an activation of a module with its control input `op(n)`, if the activated module requests a call to a function, `top` engages a sequence of activations to model the behavior of function call and return.

The Controllers `bottom1` ***and*** `bottom2` The controllers inside the modules (`bottom1` and `bottom2`) know the explicit contracts of the threads and functions. They perform an on-the-fly production of the contract of a module, from the contracts of its threads and functions. They compute an asynchronous product, restricted by the effect of event-based communication between the threads and functions.

The controllers `bottom1` and `bottom2` are thus in charge of providing the set of possible transitions of the module in response to an activation with `enq`. They also translate an input `op(n)` into the appropriate activation of a thread or function with input `op`. Finally, they route the activations `callf`$_i$ and `contf`i to the appropriate component. Conversely, they copy the con-

```
...
ⓐwhile(true){
  x++;
  e3.notify();
  e4.notify();
  ⓑwait(e1);
  if(x < 42){
    ⓒp.f(x);
    ⓓp.g(x);
  }
  y=0;
  while(y<5){
    y++;
    ⓔwait(e2);
  }
}
```

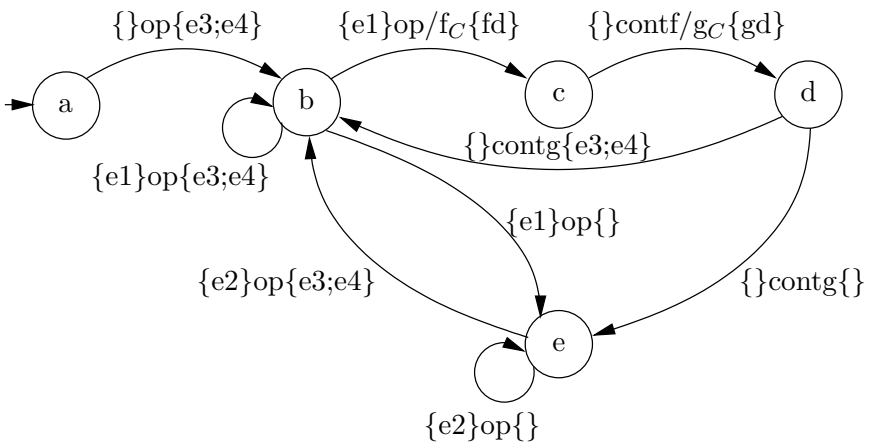


Fig. 7.16: A SystemC thread and its corresponding non-deterministic control contract

trol outputs $\texttt{endf}_i$ and $\texttt{f}_{iC}$ of the threads and functions into the corresponding outputs of the module. The details of their implementation were given in Section 6.2.2.4.

7.3.2.2 Automatic Extraction of Control Contracts from SystemC-TLM Code

To better understand the relation between 42 contracts and SystemC code, notice that contracts can be automatically *extracted* from SystemC code. Figure 7.16 gives an example SystemC thread, and the corresponding contract. The states of the contract correspond to points where the SystemC code waits for events, or calls functions (the notation ⓐ is added for the example, it is not part of SystemC). A transition `op` corresponds to a piece of code that stops on a `wait`, or on a function call. Events are translated into data dependencies. The tests on data produce non-determinism. For instance, from state ⓑ, there are three possible paths, depending on `(x < 42)` and `(y<5)`, leading to states ⓑ, ⓒ or ⓔ.

A method for extracting 42 contracts automatically from SystemC code is under development; it is based on the SystemC front-end Pinapa [MMMC05b]. Pinapa can also be used to automatically generate the architecture of the 42 model from the architecture of the SystemC model, and from an analysis of its code, to determine which events are waited for, and which functions are called.

7.3.3 Typical Uses of the Approach

The system of Figure 7.17 is made of: a DMA (Direct-Memory-Access) component, a CPU, a memory, and a bus. The behavior is as follows: the embedded software running on the CPU first writes something to the memory from address `a1` to address `a2`; then it programs the DMA to perform a transfer of this portion of the memory to another place (say, between addresses `a3` and `a4`). The advantage of using a DMA is that the CPU can then do something else while the memory transfer is performed. When it is finished, the DMA sends an interrupt to the CPU, which can repeat the same behavior.

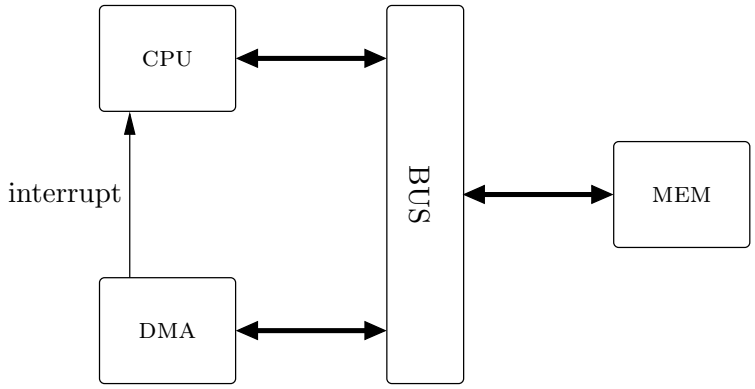


Fig. 7.17: An example of hardware architecture for *Systems-on-a-Chip*

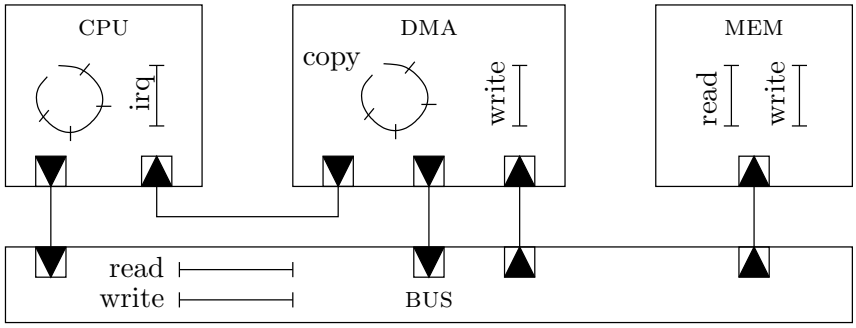


Fig. 7.18: SystemC/TLM model of the Figure 7.17

In SystemC/TLM, the model is that of Figure 7.18. The four hardware elements are components, communicating via transactions. The communications from the CPU to the memory or the DMA, and the communications between the DMA and the memory, are transactions through the bus. The interrupt is a direct transaction from the DMA to the CPU. When the CPU writes to the memory, this is modeled as follows: the thread of the CPU calls the function `write` of the bus, which itself calls the function `write` of the memory. The 42 model of Figure 7.19 is built as explained in section 7.3.1. We do not give all the details.

In the sequel we present the benefits of establishing the correspondence between SystemC-TLM and 42. First, we show how the contracts of 42 may be used to detect synchronization bugs in existing SystemC-TLM models; 42 may also be used as a starting point of the design of SystemC-TLM virtual platforms; finally, we show how models written in SystemC-TLM and models written in 42 may be mixed and executed together.

7.3.3.1 Debugging Existing SystemC-TLM Code

Suppose we are provided with an already existing virtual prototype for debugging purposes. The model is written in SystemC-TLM and is provided together with the software application running on the CPU. The code of such a platform may be quite huge[3] and hard to debug. The current approaches to debugging SystemC models rely on simulation. This may be complex because the debugging session deals with all the information of the components. Even if we only want to check the synchronization effects between the embedded software and the rest of the hardware components (e.g., the DMA), we have to observe the full simulation through carefully chosen breakpoints.

The 42-ization of the SystemC model may help detecting bugs related to synchronization be-

[3]The SystemC model of STI7200 chip contains millions LOCs

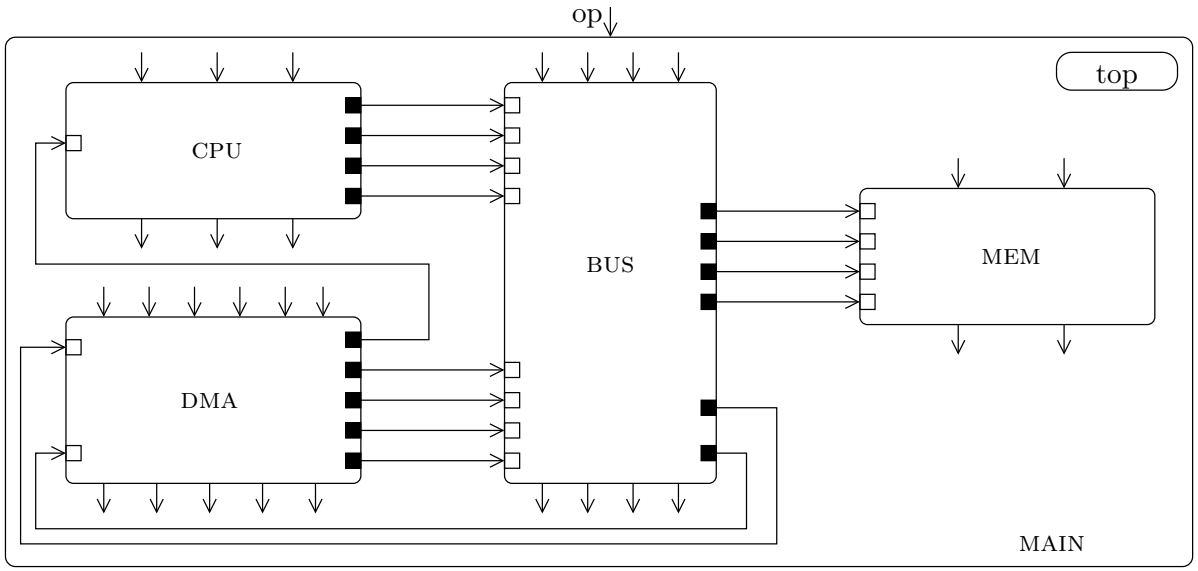


Fig. 7.19: 42 model for the system of Figure 7.18

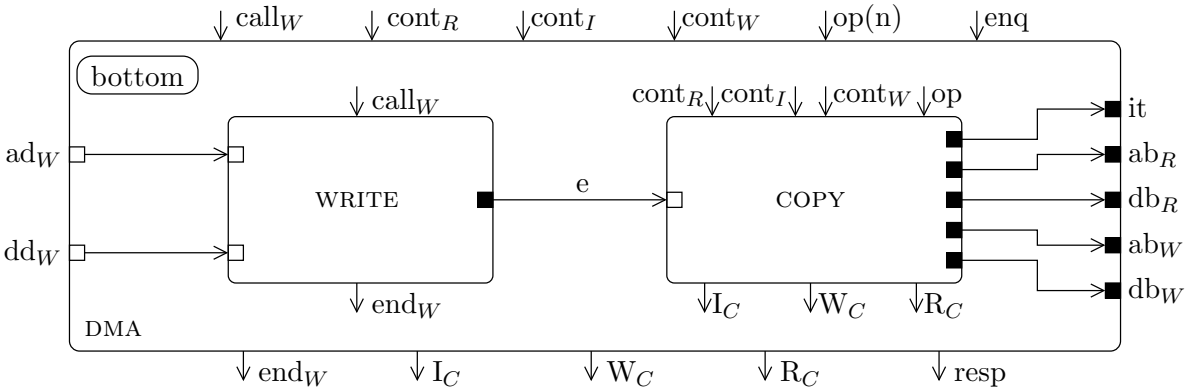


Fig. 7.20: 42 model of the DMA component

tween components. First, we extract a detailed architecture and control contracts from the SystemC code of the components as described in Section 7.3.2. For instance, for a real DMA model written in SystemC, we would extract a detailed architecture like that of Figure 7.20 (showing that the DMA is made of a function WRITE and a thread COPY).

Extracting the Contracts of the Components Once the architecture of the system is defined in 42, the contract of each function and thread of the HW components are extracted. Figure 7.21 illustrates the contracts of the thread COPY and the function WRITE of the DMA. This DMA is quite simple (only one transfer at a time, no suspension), but the SystemC code is already 50 to 100 lines.

In the contract of the thread, non-determinism comes from the abstraction of data in conditionals. State `t3` corresponds to the state of the DMA where it either terminates the whole transfer, or starts the transfer of a new word.

In the contract of the function WRITE, we see conditions on specific values of a data input (e.g., ad$_W$=08). The DMA has three local registers at offsets `00, 04, 08`, where the CPU should write the start and end addresses of the memory transfer, and the transfer command, respectively. The first two are *data* registers; the last one is a *control* register, in the sense that writing to

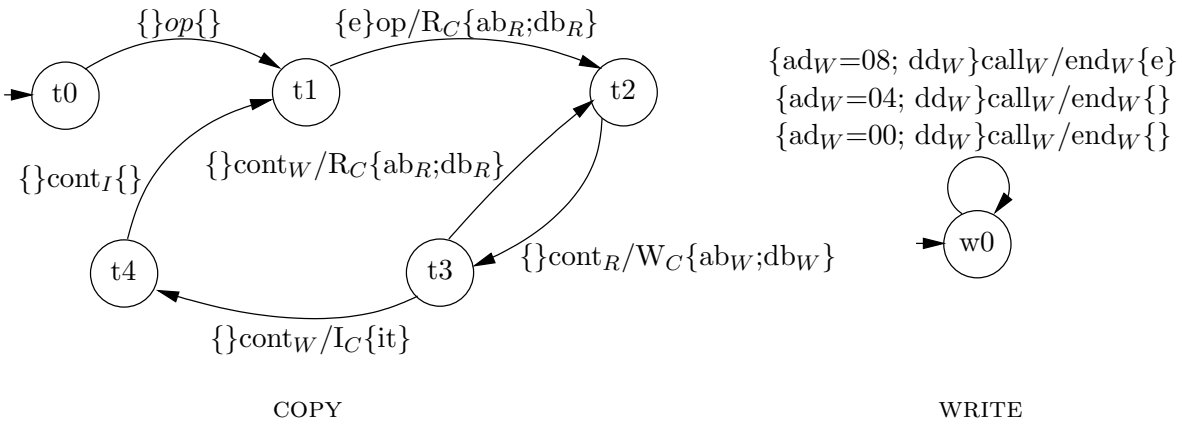


Fig. 7.21: 42 contracts of the DMA component

it triggers some behavior. This is visible in the contract of the function WRITE: if the input address is `08`, the transition produces the event `e` that triggers the thread COPY to start the transfer.

One may wonder how such a contract may be extracted automatically from a piece of SystemC code, in particular the explicit values associated with the data inputs. In most of the cases, the function is written as a `switch` statement with explicit constants for the cases, which makes the extraction feasible. In other cases, we could try to exploit the information on which registers are *control* ones, as specified in semi-formal interface definitions like IPXact [Con].

Simulation of the Contracts to Detect Synchronization Bugs Once the architecture of the system and the component contracts are extracted, we simulate the system by executing the contracts only. The simulation of contracts (in the 42 world) is lightweight comparing to the actual SystemC code. Moreover, we can define some properties to be monitored. The properties would be expressed formally in a quite elegant manner.

7.3.3.2 Using 42 Models as a Starting Point of the Design of SystemC Models

To design new SystemC-TLM platforms from scratch, 42 may be used as a starting point of the design approach. Having in mind the virtual prototype we are interested in (e.g., the system of Figure 7.17), we can design the 42 system corresponding to the future SystemC model:

- First, we identify the 42 components corresponding to the hardware blocs, and the potential communication (i.e., the transactions) between them. At this point of the design flow, we define the architecture of the system at the highest level of the hierarchy (e.g., Figure 7.19), where all the components have been connected by wires, and the required control ports have been identified.
- Second, for each component, we design the set of threads that define its active behavior, and the functions that are in charge of responding to incoming transactions. For the threads and functions, we identify those in charge of emitting transactions (if any). Also, we define the set of events that would be used to synchronize them.
- Finally, to each thread and function, we associate a control contract. As described previously, we simulate the system by executing contracts to check synchronization bugs. If we are happy, we start developing the SystemC model. Notice, that the 42 system may be used as input to some automatic tools in order to generate a skeleton of the SystemC model.

Reusing Existing SystemC Components When developing systems in SystemC/TLM, it is often the case that there exist models for usual components like the DMA, the bus, and the memory. The model of a memory is simple, but the model of a real DMA can be quite complex, because it can allow several transfers at the same time, and/or it can offer support for suspending/resuming a transfer, etc. The developer has to build a system by connecting those existing components, and then has to write embedded code to be executed by the CPU (with native simulations, or by using an ISS). Debugging the software to check if it synchronizes with the hardware components (e.g., the DMA) may be quite complex. The correspondence we have defined allows the following alternative approach:

- First, we extract the architecture and the contracts of the existing components (i.e., the DMA, the memory, the bus).
- Second, we write from scratch a non-deterministic contract for the CPU component (it is in fact a contract of the CPU, plus the embedded software that will run on it); we execute this CPU+SW contract together with the extracted contracts, in the 42 world; we see only the synchronization effects, not the general complexity of the SystemC code. Moreover, since the 42 model exposes clearly the control effects hidden in function calls, the points where the system has to be observed to debug the synchronization effects are already built in the model. The simulation speed can be quite good, also, because of the simplicity of the model. If the simulation of the contracts reveals no bugs, we may start developing the software code with respect to the contract we designed.
- Once we are happy with the contract of the CPU+SW, having played with it in the context of a complete SoC, we start developing the real embedded software (that will be executed by a SystemC component which is an instruction-set-simulator of the CPU). At the end, we may still execute the set of 42 contracts, *together* with the actual SystemC code of the CPU (which, itself, executes the embedded software). Executing SystemC code with the contracts is described below. This is a way of checking, dynamically, that the embedded software, together with the ISS, conforms to the abstract CPU+SW contract.

7.3.3.3 Mixing the Two Models

The way we established the correspondence between SystemC and 42 models, makes it easy to mix both of the execution models. That is, making C++ code (written for SystemC-TLM components) execute with 42 controllers, and executing 42 control contracts with the SystemC scheduler. The details about how we can do that, and the benefits we get are described in the sequel.

Execution of 42 Control Contracts with SystemC Scheduler For the development of new components for SystemC-TLM platforms, engineers are used to write rough implementation of the components under development to play with synchronization effects with the rest of the platform. Such an implementation may contain some non-determinism encoded with `rand` statements.

Instead of writing rough code of the components under development, we propose to implement the functions and threads of the components as 42 contracts. The 42 control contracts are well suited to describe the synchronization of components; replacing the rough implementation of a component with a 42 contract would yield a better clarity of the intended behavior of the component.

Encoding an automaton with a general programming language (e.g., C++) is not so hard. Figure 7.22 is a possible encoding of the contract of the thread COPY of Figure 7.21 in SystemC-TLM (C++). The states of the contracts are encoded with a `switch` statement, the label of each

```
int state= 0;
while (true){
  switch(state){
  case 0: state = 1; break;
  case 1: wait(e); p.R(ab_R, db_R); state = 2; break;
  case 2: p.W(ab_W, db_W); state = 3; break;
  case 3: if(rand()%2)
            p.R(ab_R, db_R); state = 2; break;
          else
            p2.I_c(it); state = 4; break;
  case 4: state = 1; break;
  }
}
```

Fig. 7.22: An encoding of the contract of the thread COPY of Figure 7.21 in SystemC-TLM

transition is translated into the corresponding SystemC code (i.e., function calls and waiting for events).

Execution of SystemC Code with 42 Controllers The principles of executing the SystemC code with 42 controllers are as follows. A SystemC module can be compiled separately, yielding an object code with entry points for threads and functions. Each 42 component corresponding to a thread or function is a Java wrapper for the corresponding entry point in this object code, connected to it via a JNI interface. This execution mode is similar to what we described in Section 7.2.1.

We provide re-implementations of the `event` class, and of the `wait` and `notify` methods. Each time a piece of SystemC code calls a `wait (e)`, or a `e.notify()` this is intercepted by the wrapper, so that the 42 world knows about the events exchanged. We also have to intercept the function calls. This is done by re-implementing the class `port` of SystemC.

When a 42 component wrapping a SystemC thread or function is activated, the wrapper (i.e. the component) lets the SystemC thread or function execute, until it reaches a `wait` statement or a function call, which is intercepted by the wrapper. The wrapper will report on the activity of the SystemC code, and yield the control to the 42 controller. The control will not return to the SystemC code until the next activation of the wrapper.

Matching the State of the Code with the State of the Contract To execute the code of a function or thread together with its corresponding 42 control contract, we had to solve the general problem that appears when executing *non-deterministic* contracts together with *deterministic* code: how to establish the correspondence between the states of the contracts, and the control point in the actual execution? A solution to this intrinsic problem exists when the implementation (here, the SystemC code) is written by somebody who knows the contract; the programmer has to relate explicitly the code he/she writes to the states of the contract. This can be done by using a special annotation function `state`. Figure 7.23 is an example implementation of a contract, where the correspondence is made (the initial state does not need to be specified with a `state` annotation).

The special function `state` we introduced is intercepted by the wrapper, which can inform the 42 controllers (that interpret contracts) about the current control point in the SystemC code.

If we cannot rely on these annotations, there is no way to extract the correspondence automatically, even with sophisticated program analysis techniques. The points corresponding to states are easy to discover, but we cannot decide which of the contract states they correspond to.

```
while(true){
  x++;
  e3.notify();
  e4.notify();
  state(b) ;
  wait(e1);
  if(x < 42){
     state(c);p.f(x);
     state(d);p.g(x);
  }
  y=0;
  while(y<5){
    y++; state(e);
    wait(e2);
  }
}
```

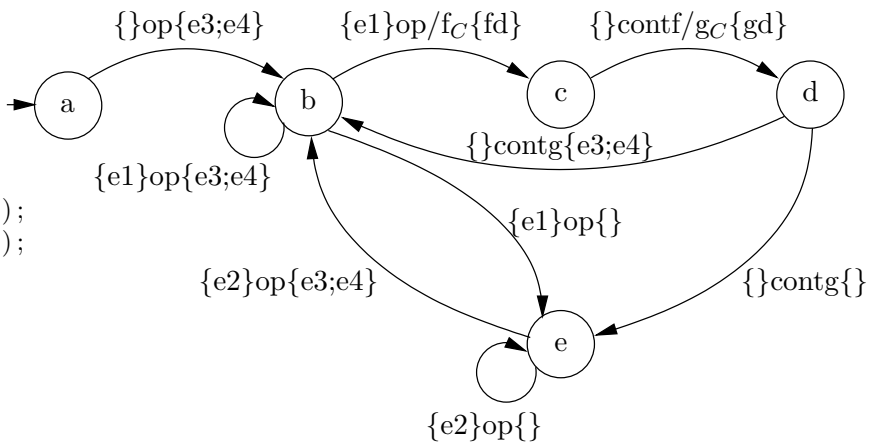


Fig. 7.23: Annotations in SystemC code to match the state of the contract

7.3.4 Comments

Extending the model with time and shared variables. In the example, we did not use *time*, nor *shared variables*. However, the example is sufficient to illustrate the key points of the correspondence with 42. We may generalize our correspondence scheme to allow shared variables in a module, and timed models. This is a bit more complex than the version presented here, but follows the same lines. The shared variables would be modeled as components; the notion of time would be an additional decoration of the transitions in a contract. The controller on the top of the hierarchy will play the role of a global time keeper.

The complexity exposed by the 42 model is intrinsic to formal models. The correspondence between SystemC/TLM and 42 may seem complex. In particular, there are more ports and connections between components in the 42 version, and mimicking the effect of the SystemC scheduler with two levels of controllers is more complex than flattening the structure and relying on function calls, as in SystemC. However, we think 42 should not be blamed for that; it only makes explicit the complex synchronization patterns of TLM components that are hidden in the SystemC execution engine. In other words, the complexity of a component in its 42 version is a better representation of its intrinsic complexity, than the SystemC/TLM version.

Reusing components requires detailed description. The 42 version exposes exactly the information that a user should have in mind when trying to reuse a TLM component. If we want to use SystemC/TLM modules in other contexts (42 or another one), we need to *expose* functions offered to the other modules, including the control dependencies they imply. Conversely, the two levels of controllers allow to make the effect of events as *locally hidden* as it should be; this makes it possible to use TLM modules written in other languages, with a local managing of internal synchronization, with events or another mechanism.

False positive errors. The 42 contracts are some abstraction of the behavior of the threads and functions of a SystemC component. This means that their execution exposes more behaviors than the execution of the SystemC code would do. Hence, the execution of contracts may detect some bugs which may be *false-positives*; that is, errors that will not appear in the real SystemC model. Though, one can check if an error is indeed

observable in the SystemC model: we just execute the SystemC code together with the control contracts and reproduce the execution trace that led to the error.

On the formalization of TLM. There has been a lot of work on the formalization of SystemC, as a parallel programming language. In [NH06, PS08], the approach is to write formal models by hand, by reengineering a *System-on-a-Chip* description in SystemC; this allows clever abstractions, but forbids the direct use of SystemC models. Another idea is to extract formal models automatically from SystemC programs, so that tools like model-checkers can be applied. In [HFG08], the formalization uses timed automata, with a connection to Uppaal [LPY97]. [MMMC05a, MMC+08] describe several formalizations and connections to SMV [Mcm92a], SPIN [Hol97], etc. However, all these formalizations include some form of a *global* model for the SystemC scheduler, and the TLM principles in SystemC are not formalized, in the sense that there is no clear definition of what a *TLM component* is. *Guidelines* only exist for the developers of TL models.

Chapter 8

Related Work

Introduction (En) *The 42 model tackles some notions in relation with the modeling of heterogeneous embedded systems. In particular, it addresses the notions of components, contracts, and MoCCs. A lot of work has been dedicated to the design and modeling of embedded systems, components and specification languages. Some of the related work were already presented in Chapter 2. This chapter presents other approaches related to 42, not all of them are dedicated to embedded systems. We cannot be exhaustive and give a complete list of the approaches, we present some of them in order to discuss the choices we made for 42.*

Contents

Introduction (Fr) *42 est une approche à composants pour la modélisation des systèmes embarqués hétérogènes. Durant la présentation de cette approche, nous avons vu que 42 s'intéresse particulièrement aux notions de composant, contrat et de* MoCC*. Beaucoup de travaux ont été dédiés à la conception et à la modélisation des systèmes embarqués, ainsi qu'au développement de modèles à composants et de langages de spécifications. Dans le chapitre 2 nous avons abordé quelques approches existantes. Les travaux connexes à 42 sont nombreux. Nous ne pouvons pas être exhaustif et citer toutes les approches. Dans le chapitre actuel, nous essayons de compléter la liste des travaux connexes par quelques approches afin de discuter les choix que nous avons adopté pour 42. Ces approches ne sont pas toutes dédiées aux systèmes embarqués.*

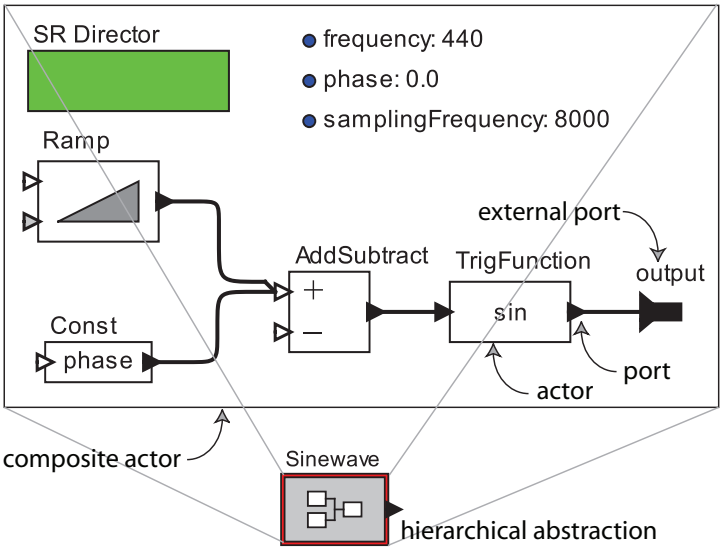


Fig. 8.1: The design of a composite actor in Ptolemy

8.1 Component Models and *MoCCs*

This section describes some component-based modeling approaches and comments on their relation with 42. We dedicate a significant part of this section to the description of Ptolemy, since 42 is mostly inspired from this approach (Section 8.1.1). Then, in Section 8.1.2 we discuss the expressiveness and design choices of 42 regarding to modeling approaches of embedded systems in general, and the Ptolemy approach in particular. Sections 8.1.3, 8.1.4, 8.1.5 are dedicated to reactive modules, Fractal and Reo respectively.

8.1.1 Ptolemy

Ptolemy [EJL+03] is a component-based model designed for the purpose of modeling heterogeneous embedded systems. It is equipped with a graphical simulation tool that allows for combining several *MoCCs* hierarchically.

Figure 8.1 [1] illustrates the modeling of a system in Ptolemy. The basic blocks (`Ramp`, `Const`, etc.) to build a system are *actors* (in the sense of [HBS73]) which have input and output ports. Each system is associated with a *MoCC* (a *MoCC* is called a *domain*) and is implemented by a *director*. The assembly of Figure 8.1 is associated with the `SR (Synchronous Reactive) director`. The role of a director is to manage the scheduling of actors.

Actors are connected with *relations* (the wires), through which they communicate tokens. The set of relations between actors define the architecture of the system. Ptolemy is hierarchic, a set of connected actors and a director may be encapsulated in order to form a *composite actor*. The composite actor also exposes new input and output ports. The component `Sinewave` of the figure is the encapsulation of the assembly.

Actors An actor in Ptolemy is an entity having ports from which it may consume and produce tokens. A port may be an input port, an output port, or an input/output port. An actor reacts when the director *fires* (activates) it.

[1] The figure is borrowed from [LZ07]

Each port may be associated with some parameters such as the multiplicity, and the consumption (resp., production) rate. A port allows one relation to be connected (multiplicity equals to one) to it, while a multi-port allows more than one relation. The rate parameter defines how much tokens would be consumed or produced at each firing.

To be used by a director, an actor should implement some methods related to its initialization, execution, and finalization. We focus on the *executable* interface that an actor must implement. This interface includes the following methods:

prefire() is called prior to the method `fire()`. It checks whether the preconditions of the firing of the actor are satisfied or not. For instance, it may return `false`, if some of the inputs of the actor do not have enough tokens.

fire() may be called if `prefire()` returns `true`. The effect of such a method is to compute the output tokens depending on the input tokens, and the internal state of the actor. It should not modify the state of the actor.

postfire() is called to modify the internal state of the actor. To this end, it may read the input tokens.

An actor is called *domain-polymorphic* when it implements these methods as they were described; in addition, it guarantees them to be finite. A domain-polymorphic actors may be used in an assembly, whatever the *MoCC* implemented by the director.

However, some actors may not guarantee such a behavior, thus they may be used just with particular *MoCCs*; they are *domain-specific* actors. For instance, some actors may not guarantee the state to be unchanged when the `fire()` method is called; others may not guarantee the methods to be finite.

Directors Ptolemy provide several directors organized in a catalogue. In the sequel we present some of them to expose some details about the possible interactions between an actor an the directors:

Process Network This domain is an implementation of the Khan Process Network [Kah74, KB77]. Within this domain [Goe98], each actor is associated with a thread which continuously calls the methods `prefire()`, `fire()`, and `postfire()` of the actor. Actors execute concurrently, reading from an input is made blocking with the PN domain.

Data Flow In this domain, actors are not associated with threads, they are fired directly by the director depending on a scheduling policy. The scheduling may be computed statically as it is done in the *Synchronous Data Flow* domain which assumes fixed consumption/production rates. In the *Dynamic Data Flow* domain [Zho05], the scheduling is computed at the beginning of each iteration because the rates may change from an iteration to another.

Continuous Time This domain deals with continuous time [Liu98], where the director manages a global notion of time. It advances the time by fixed steps, small enough to approximate the continuous behavior. The actors are fired at each step.

Discrete Events In this domain [Lee99], the tokens communicated between components are associated with a time stamp; and are maintained in a queue. The director retrieves the events from the queue starting from the oldest ones, and fires the actor to which these tokens are intended. During the firing, the actor may compute some tokens to which it associates a time stamp. Actors have also the possibility to request a firing at a given point in time.

A director may impose on the actors to be polymorphic. That is, they should implement the executable interface strictly as it was described previously. Others may impose less constrained implementation but still they may use polymorphic actors.

Through the interface of the composite actor, the encapsulated director exposes the executable interface. Depending on its inner director, a composite actor may or may not be domain-polymorphic. Thus it may not be used in any domain. This raises the issue of *MoCCs* compatibility when combining them.

***Combining* MoCCs** In Ptolemy, directors are combined hierarchically to model heterogeneity. Some *MoCCs* may be combined without problem, and others are incompatible. The incompatibility raises when the inner director of a composite actor may not expose the required executable interface required by the domain in which the composite actor is used. Also, incompatibility raises because the notion of time. Directors may deal with the notion of time differently, which may cause some issues related to time resolution when combing them.

Comments

Predefined Vs Programmable MoCCs Ptolemy provides an extensible catalog of predefined directors, each one implementing a *MoCC*. The equivalent notion of directors in 42 is that of controllers. Our point of view is that, if we need to model all possible types of structured interactions, we should not rely on a catalogue of predefined interactions, but we should be able to program new interactions in the model. In 42 the *MoCCs* are implemented by controllers as small programs that express several types of concurrency and structured interactions in terms of basic primitives.

Allowing programmable *MoCCs* has the benefit of clearly describing the concurrency between components. The operational description of the *MoCC* has something to do with that. Moreover, the way controllers are written in 42 makes it explicit the exchange of information between components and the controller. Such information is required by the controller to structure component interactions. This is something which is not often clear in Ptolemy.

The most relevant work around *MoCCs* is the family of TAG semantics [BCCSV05]. In some sense, 42 is an intermediate point of view, between the way MoCCs are programmed but not fully formalized in Ptolemy, and the way they are formalized but far from programming purposes in the TAG semantics.

Expressing parallelism In Ptolemy, components may execute in parallel. For instance, in the KPN *MoCC* as implemented by Ptolemy, components are run as threads with blocking read from the inputs. In 42, we did not provide a semantics of parallel activation of components. The controller activates the components in sequence. When we need to express parallelism, we rely on interleaving their activations. This provides a better means for reasoning than parallel threads would provide.

Components and Contracts The `fire()` method of a Ptolemy actor is the unique entry-point to make the actor perform some computation. Components in 42 may be associated with several entry-points (i.e., control inputs) through which they may be activated. At each activation Ptolemy actors may require a certain amount of input tokens on the same input port, while 42 components use one value from each input at each activation. This also holds for output ports.

The catalogue of Ptolemy actors may be extended with new actors. For an actor to be domain-polymorphic, its implementation of the executable interface (i.e., `prefire,`

`fire, postfire` methods) should follow the description given above. This constitute an implicit contract for a domain-polymorphic actor. Other contracts are derived by relaxing some of the constraints [GBA+09]. The usability of an actor within a *MoCC* relies on such implicit contracts. With 42, beside implicit contracts, components may be described with explicit ones. Even if a component is designed for a particular *MoCC*, it may be used with another *MoCC* as long as the controller is consistent with the component contract.

8.1.2 General Discussions on Design and Expressiveness of Models

This section discusses some points related to the expressiveness and design choices of 42 regarding to some existing modeling approaches for embedded systems.

Describing various MoCCs within the same model The design of embedded systems involves some expertise from various domains. In each domain there exist some well defined models that fit the requirements. We think that a modeling framework for embedded systems should be able to combine such domain specific models in order to provide a global model for an embedded system. This means that a modeling framework should be able to describe various *MoCCs* within the same formalism.

When the modeling framework is associated with a particular *MoCC*, as it would be the case for synchronous languages [HCRP91, BG92, GG87], it is often the case that the designer uses some languages features to describe other concurrency models [HB02].

As in Ptolemy, 42 belongs to the family of models where the *MoCC* is part of the design not a built-in notion of the language. Other approaches follow the same point of view, see for instance ModHel'X [HB08], Rialto [BL02], etc.

Architecture Description Languages and *MoCCs* In 42, the ADL has a dataflow style. Since the connections have no meaning until the *MoCC* is defined as a controller, this only means that we express data dependencies explicitly in the ADL, and control aspects in the *MoCC*. For 42, we chose only one style of architecture description language, because we want a simple formal semantics.

In Ptolemy, the notion of *MoCC* is somewhat extreme: given a picture made of boxes and arrows, it is possible to consider it, either as a dataflow diagram, or as an automaton, just by changing *MoCCs*. This means that even the interpretation of the architecture-description (ADL) part is left to the *MoCC*: the ADL, used to group components at a given level of the hierarchy, may be dataflow (in which case the components are implicitly in parallel), or given as an explicit automaton (in which case the components execute sequentially), or anything else that could be expressed in a new *MoCC*. This is a key point in Ptolemy for building the family of *modal models* [LZ05], in which an automaton is used to control several activities associated with its states, and described with other *MoCCs*.

Oriented connections vs non-oriented ones 42 adopts a dataflow style architecture description language, with oriented connections. In Ptolemy, in the modeling tool Spice [SPI] for electronic circuits, or in the bond graph formalism [Tho75], this is not necessarily the case, allowing the modeling of various physical behaviors. With 42 we concentrate on discrete computer systems.

Even for modeling computer behaviors, some models choose symmetric synchronization primitives like rendez-vous, thus relying on non-oriented connections (see for instance the *Architectural Interaction Diagrams*, or AIDs [RC03])

Strict Hierarchy A basic component, or a composed component built as an assembly of other components, are perfectly undistinguishable in any 42 context. This is true also for Ptolemy, Fractal [BCL+06a], and to some extent SystemC-TLM [Ghe06]. Early version of the component model BIP [BBS06] did not have a dedicated notion of encapsulation that could hide the details of an assembly and allow to consider it as a basic component, but this was fixed in recent work.

In some formalisms developed in the architecture description languages community, there is also a clear distinction between the set of elementary components, and the object obtained by combining them with an ADL. However, the focus is more on component interactions [AAAG+05, SDK+95a, SDK+95b].

We consider this strict hierarchy property to be a key property of component-based frameworks, because it allows to forget as much as possible about the details of the components, as soon as possible. Moreover, the hierarchy is essential for the modeling of heterogeneity, since we do not allow to use several MoCCs at the same level.

Continuous vs discrete models 42 is limited to the discrete case. When we need to include the physical environment in a model, we can consider components that are non-deterministic discretized versions of some continuous models, but we do not study how to mix continuous and discrete MoCCs. Ptolemy addresses this problem. Moreover, it allows the combination of discrete and continuous models in the same system description, which is really useful for embedded systems that may include digital and analog parts.

Other proposals, like VHDL-AMS [vhd99] or SystemC-AMS [VPB+08] concern the modeling of mixed digital-analog systems, but they do not address the component aspects. Moreover, they concentrate on the collaboration between a numerical solver and a discrete simulation engine, from a quite operational point of view, without trying to define the *semantics* of this heterogeneous combination. Similarly, Matlab/Simulink designs can mix continuous and discrete parts, but the notion of a component is not dealt with specifically. In both cases, if the collaboration between a numerical solver and a discrete simulation engine involves a fixed-step sampling of the continuous part, the result can be expressed easily in a discrete framework, where one of the components is a discretized version of a continuous object; this is what we provide with 42. Nevertheless, there is a need for mixed discrete/continuous models, but the problem is the semantics, not the implementation.

8.1.3 Modeling Synchronous/Asynchronous Systems with Reactive Modules

Reactive modules [AH99] is a formal approach to modeling discrete concurrent systems. This approach aims at modeling synchronous and asynchronous systems in the same formalism. The goal is to allow for hardware/software codesign and verification.

The modeling language proposed by this approach allows to construct models hierarchically by composing non-deterministic modules in parallel. Modules may be seen as components and have a clear separation between the internal state and the interface variables.

The Modules Figure 8.2 illustrates a reactive module (named *latch*). Such a module would be used as a basic block for describing hierarchical systems. A module consists of a collection of *variables*, and of set of *atoms* that modify the values of the variables in discrete *rounds*.

The Variables The variables are organized in three categories: *external* variables (similar to inputs) are modified by the environment (e.g., `set, reset`); *interface* variables (similar to outputs) are modified by the module (e.g, `out`); *private* variables are modified by the module

```
module latch
  external set, reset: B
  interface out: B
  private state: B;
  atom out reads state
    update
      | true → out' := state;
  atom state awaits set, reset, out
    init update
      | set' = 0 ∧ reset = 0 → state' := out'
      | set' = 1 → state' := 1
      | reset' = 1 → state' := 0
```

Fig. 8.2: A reactive module modeling a synchronous latch

and are not observable by the environment (e.g., `state`). In a reactive module, the value of a variable at the beginning of a round is referred to by unprimed identifier (e.g., `reset`); and to its value at the end of the round with primed identifier (e.g., `set'`).

The Atoms An atom declares a set of guarded-actions that are executed during the initialization round (`init` keyword), or during the update round (`update` keyword), or both (`init update`). During a round, if not stated explicitly, at most one action with a satisfied guard of an atom is executed. If more than one action is possible, a non-deterministic choice is performed.

When several modules are connected together (connection is performed by renaming the variables of the modules), all the atoms are run in parallel to execute one of their actions. However, there may be an order in which actions of the atoms are executed. For instance, the expression **atom state awaits set, reset, out** declares an atom that modify the variable `state`, and should be executed after the variables `set, reset, out` have been given new values. This creates a order of execution between this atom and the other atom of the module, because the latter is in charge of updating the value of the variable `out`.

Modeling Heterogeneity with Reactive Modules In reactive modules, the module is in charge of defining whether it behaves as a synchronous module or as an asynchronous one. An asynchronous module is a module that may leave the values of some of its interface variables unchanged during a round. This is possible because an atom may declare some empty action with a guard `true`. Hence, it is possible to take this action at any round. If a module is not asynchronous, it is synchronous.

In reactive modules, it is possible to apply some operators on a synchronous (resp., asynchronous) module in order to force it to behave as an asynchronous (resp., synchronous) one. For instance, consider the expression **next Y for P** where **next** is an operator, `Y` is the set of interface variables, `P` is an asynchronous process. The operator **next** forces the process to execute in the same round, as many actions as possible to update the values of all interface variables. Thus, such a process would behave as a synchronous one following the definition of synchrony of reactive modules.

comments

Non-determinism A modeling framework for embedded systems should allow for describing non-deterministic behaviors. Allowing for non-determinism means that components may be described with an abstract view of their behavior. This because some components may not be known, or provided as a specification only.

Reactive modules and 42 allows for describing non-deterministic behaviors. At a given round, a reactive module may have more than one actions to perform. Similarly, a 42 component may be non deterministic. Moreover, we allow controllers to embed some non-determinism when activating components. See for instance, contract interpreters in Section 6.2.2

Modeling heterogeneity Modeling heterogeneity with reactive modules relies on adapting the behavior of a component so that it fits in the desired computational model. Such an adaptation yields a new module. We gave an example of how to make an asynchronous module behave as a synchronous one. In 42, as in Ptolemy, we model heterogeneity by putting distinct *MoCCs* in a hierarchy. The concurrency model is implemented by the controllers, which allows to use components within a particular *MoCC* even if they were not designed for it.

Separation of implementation and specification In reactive modules, an atom declaration describes its data dependencies. Based on these information, the modules are scheduled in order to expose the expected behavior. In 42, this would be something related to the specification part. The contract of a component describes its data dependencies. In 42, the emphasis is put on the encapsulation aspect which requires specification, it is not the case with reactive modules.

8.1.4 An Academic Approach to Software Components: Fractal

Fractal [BCL+06a] is a component-based model designed for the development of systems and applications ranging from operating systems to graphical interfaces. It makes a strict separation between the interface of a component and its implementation. The communication between components is well detailed in order to make the software architecture explicit.

Fractal is not tied to a specific language, many frameworks implementing Fractal specifications were designed. Beside Julia [BCL+06b], its reference Java implementation, many frameworks exist such as Think [FSLM02], a C-implementation for operating systems development, FracNet [SPED06], a .NET implementation; etc.

Hence, Fractal is provided as a specification that defines the notion of components, and their composition. It also defines the set of non-functional interfaces a component should implement, in order to be used in framework based on the Fractal specifications. Such a framework would allow instantiating systems, and has many features related to components reconfiguration (at launch-time or at runtime); component introspection, life cycle management, etc.

Components Figure 8.3 is a system designed with Fractal. The system `ClientServer` is composed of the component `Client` and the component `Server`. Fractal is hierarchic, the `ClientServer` is also a component and may be used in other designs.

A Fractal component implements two types of interfaces: *functional* and *control* interfaces.

- A functional interface is an interface that corresponds to a provided or required functionality of a component. The component `client` has a provided interface `m` of type `M`, and a required one `s` of type `S`.
- A control interface (e.g., `C`, `BC`, etc.) defines a set of methods related to the non-functional aspect of a component such as introspection, reconfiguration, life cycle management, etc. For instance, `BC` (Binding-controller) is an interface that allows to connect a functional interface of a component to a functional interface of another component.

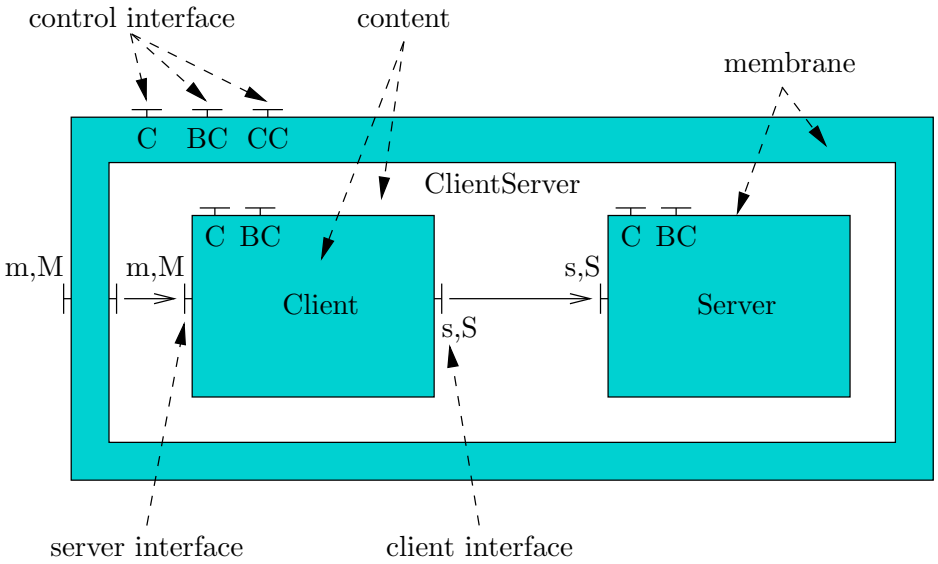


Fig. 8.3: The design of a Client/Server system in Fractal

A Fractal component is composed of two parts, a *content* and a *membrane* (also called a controller). The content part defines its functional behavior, it describes the implementation of the component. The membrane embodies the behavior of the component, its consists of the set of control interfaces the component may have.

Assembling Components Fractal components are connected with *bindings*. A binding may connect a required functional interface of a component to a provided functional interface of another component. For instance, the required interface `s` of the `Client` is connected to the provided interface `s` of the `Server`. A binding may also connect a provided (resp., required) interface of the encapsulating component to the provided (resp., required) interface of an inner component. The connection between the interface of `ClientServer` and the interface of `Client` is an example.

The bindings make the architecture explicit. They describe how the control and data flows pass through the components. For instance, when a method is called through the interface of the component `ClientServer`, the call is transported to the interface of the component `Client`. On its turn, the `Client` component may call a method through the provided interface of the component `Server`.

In Fractal, there is no controller to define the activation of components. Components communicate through function calls.

Comments

Components interaction Fractal components communicate via service calls. The control flow together with the parameters are passed to the callee at the moment of the call. In 42, it is the responsibility of the controller to manage the activation of components and their communication.

Controllers Vs Control interface In Fractal a control interface (i.e., the membrane) wraps each component. The membrane intercepts the service calls at the border of the composite component and translates it into a call to a subcomponent service. The membrane of a composite component corresponds to the controller of 42 components. However, a controller is global to a set of components, while the membrane may be local to a component.

The membrane manages some non-functional aspects such as: life-cycle management, reconfiguration, introspection, etc. The controllers in 42 are, in general, concerned with the *MoCC*. Still, they may manage some non-functional aspects such as introspection (see Section 6.2.2).

Runtime configuration Runtime configuration is an important feature of software component models. Fractal supports such a feature, allowing for stopping components at runtime and replacing them by other ones. We think that runtime configuration is not so important for embedded systems, in particular for critical systems. Imagine what would happen if we replace part of the software during the flight of an air plane. Moreover, the main concern of 42 is modeling, while Fractal is dedicated to component-based design of complex systems.

8.1.5 Coordination of Component Activities with Reo

The approach adopted by *Reo* [Arb04] is based on the separation between the computation performed by individual components and the communication that holds between them. While the implementation of components is left to some programming languages, the emphasis with Reo is on the connection and the communication between them.

Given a set of components of a system, Reo aims at filling the gap when interfacing them. Reo allows to describe the coordination between the components by means of connectors. The connectors form the required *glue code* to make the entities of the system interact in a desirable manner; because the correctness of individual entities does not imply the correctness of their composition.

Components Any active entity that performs computation is considered as a component. For instance, threads, modules, processes, etc., are components. The synchronizations and communications that happen inside a component are irrelevant for Reo. What is of interest is the inter-components communication that takes place through channels.

Components are supposed to execute in potentially distinct logical/physical devices. Thus, Reo may be used to describe the coordination between several threads running on a single processor, or large scale applications distributed over a network of computers. For instance, in [MA07] web-services are modeled by *constraint automata* [BSAR06], while Reo is used to describe their coordination.

Connectors Reo connectors consist of a set of communication channels. Each channel has two directed *ends*, and exposes a simple communication pattern like synchronous, asynchronous, etc., communication. A set of channels assembled in a particular topology form a particular connector. This connector is then used to connect components (see Figure 8.4), and describes complex communication patterns between them.

Related Approaches Reo belongs to the family of coordination languages first introduced by the coordination language Linda [CG89, ACG86]. Several approaches to coordination languages have been defined. Existing coordination models and languages where described in [PA98]; the authors argue that these models are mainly classified into two classes: *data-driven models* (e.g., Linda, Laura [Tol96], Sonia [Ban96], etc.) and *control-driven models* (e.g., Darwin/Regis [MDK94], Rapide [SDK+95a], etc.).

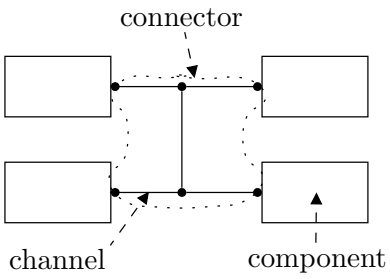


Fig. 8.4: Components connected with Reo connectors

8.1.5.1 Comments

Do connections express some behavior? Reo and the similar approaches focus on the connection between components. Connectors express complex communication patterns between components and are built by combining other connectors. Basic connectors describe simple communication patterns like point-to-point communication, broadcasting, synchronous/asynchronous communication, etc. Reo connectors are associated with some behavior and memory. A connector may memorize a data, duplicate it, or even lose it.

In 42, the connections only express that some information may flow from one component to another. There is no synchronization nor memory attached to the connections, *a priori*. The controller may decide to manage some temporary memory corresponding to the wires in order to describe complex communication patterns, but this does not mean that the wire behaves as memory for the connected components, since the lifetime of this memory is limited to the macro-step. In 42, the need for complex communications patterns usually leads to the following solution: if a communication pattern has a complex behavior, it is a component, not a connection.

8.2 Specification Languages and Contracts

8.2.1 Formal Specification of Behaviors

The work around specification languages is motivated by the fact that component-based development should rely on formal description of components. Formal approaches are used in order to develop techniques and tools that would help designing correct component-based systems.

The ancestors of formal specification languages are Process Algebra [Mil80, Hoa78], first introduced for the modeling of concurrent systems. Process algebra allow for describing the behavior of processes and provide several constructs for describing their sequential/parallel composition, communication, synchronization, etc.

In the area of component-based development, many specification languages where designed [PV02, VVR06, FS02] and where successfully ported to existing component-based models such in Fractal [BCL+06a], and SOFA [PV02].

8.2.1.1 Interface Automata

An interface automaton [dAH01] is an automaton-based description of a component. It is an automaton whose transitions are labeled with an action. The action may be an *input action*, *output action*, or an *internal action*. At each state, it expresses the *assumptions* about the environment in terms of the accepted inputs; and the *guarantees* about the behavior of the component in terms of output actions.

Interface automata are syntactically similar to I/O automata [LT87], but they impose some assumptions on the behavior of the environment. An interface automaton describes the behavior of a component only under environments satisfying the assumptions.

Composing Interface Automata In interface automata, communication between components is assumed to be directed and point-to-point. Composing two components that may coincide in some inputs and outputs (we will refer to these inputs and outputs as shared actions) yield a new component. The behavior of the new component would be described by an interface automaton computed with respect to the interface automata of its constituents.

The composition operator is simply an asynchronous product of automata with synchronizations on the shared actions. That is, at a given state of the product:

- actions of the constituents are interleaved if they are not shared.
- constituents perform one step synchronously if their transition is labeled with the same action with distinct directions (input/output).

Compatibility Issues When composing interfaces, some *illegal states* may appear in the product of interfaces. Illegal states are states where one component outputs a shared action while the other does not accept it. That is, they do not synchronize on shared actions at such a state.

However, the composition of interface automata follows an *optimistic* approach. Illegal states in an open system does not imply that the composed components are incompatible, because there may be an environment in which these states are made unreachable. Illegal states in a closed system are synonym of incompatibility.

Refinement Interface automata also tackles the refinement relation between components. An interface $\mathcal{I}'$ refines the interface $\mathcal{I}$ guarantees that $\mathcal{I}'$ may be used in whatever assembly $\mathcal{I}$ is used. Refinement acts contravariantly on input assumptions and output guarantees. The interface $\mathcal{I}'$ should have more relaxed (resp., restricted) assumptions on the environment (resp., guarantees).

8.2.1.2 Modal Specifications

Modal specifications [Lar90] describe the behavior of processes by means of labeled transition systems. The transitions are labeled with an action of the process and associated with a modality (*must* or *may*). Modalities impose restrictions on the transitions of possible implementation by telling which transitions are necessary (*must*) and which are admissible (*may*). Modal specifications extend Process Algebra in the sense that specifications may be combined using process constructs.

A modal specification may be associated with several implementations. A *must* transition is available in every component that implements the modal specification; while a *may* transition needs to be. Implementations are prefix-closed languages or deterministic automata.

Modal specifications admit refinements; a specification refines another if it preserves the required transitions, while it may restrict the admissible ones. The refinement process of a specification allows for getting the final implementation gradually.

Beside the refinement relation, modal interfaces admit composition, conjunction, and quotient.

8.2.1.3 Modal Interface

Modal interface [RBB+09] are a combination of modal specifications and interface automata. This approach extends modal specifications by typing the actions with input and output labels, à la interface automata. It then benefits from the properties of both of the two approaches described above.

8.2.1.4 Comments

Expressing modalities An action labeling a transition of a modal specification corresponds to a piece of code a 42 component would execute during an activation. Hence, every transition in a 42 contract corresponds to an action of a modal specification. However, there are no modalities in 42 contracts. If a contract declares that a transition is possible, then a component consistent with that contract must implement such a transition.

Expressing the type of inputs/outputs Interface automata allow for expressing the sequences of inputs/outputs of a component. Depending on the modeling purpose, the inputs and outputs may be function calls, events, etc. In 42, the separation between control and data and the way they are described in the contracts make a clear description of what is being modeled.

Refinement Refinement is dealt with in interface automata together with modal specifications. We think that a specification language should provide a refinement relation, so that it allows for gradually obtaining an implementation for a given specification. However, the only notion of refinement relation defined for 42 is the consistency relation between a component and its contract in Section 5.3.1.

Compatibility The compatibility between components is given a clear semantics in interface automata. This because the concurrency model between components is given a fixed definition. It is an asynchronous model with synchronization in shared actions. In 42, this is not the case. We allow for the description of several *MoCCs*. We would describe several notions of compatibility between components, each one related to a particular *MoCC*.

8.2.2 Contracts for Hardware Components: The Don't Care Conditions

For hardware components, *don't care conditions* [BBS98, Dev91, DM93, DMN90] are some specifications associated with parts of a logic circuit. Don't care conditions arise during the design phase. They play an important role in specification and optimization of the logic circuit being designed.

Given a logic component with inputs and outputs, the designer may declare that he does not care about a provided output under a certain condition. For instance, the expression `o = -` in VHDL allows for declaring that the value assigned to `o` is not of interest. Moreover, the designer is able to declare that particular configurations of inputs never occur. Such declarations constitute explicit specifications from which implicit ones may be derived.

Specifying don't care conditions depends on the abstraction level used to describe the circuit. By declaring don't care conditions, the designer allows for the synthesis of an optimized logic circuit where the network of logic gates related to don't cares has been removed.

The 42 contracts may also be used in order to express don't care conditions for logic components. However, we would used these contracts only for specifications. The aim of 42 is modeling, and is not tied to a particular design approach where sophisticated techniques like optimizations are needed.

Chapter 9

A Tool for the 42 Component Model

Introduction (En) *The 42 model that has been described so far is not associated with any particular language. This chapter presents our prototype tool implemented in Java. The tool has been used to enable the simulation of the various examples presented in this thesis. The purpose of such a presentation is to expose the basic features a 42-based tool would have. We first present how basic components are written and how we describe the architecture of systems in order to design composed components. Then we present the instantiation mechanism together with the simulation engine and its graphical interface.*

Contents

Introduction (Fr) *Le modèle à composants 42 est une approche indépendante de tout langage de programmation ou de formalisme pour la modélisation des systèmes embarqués. Dans ce chapitre, nous présentons un outil que nous avons développé en Java qui implémente les différentes notions du modèle 42. Cet outil a été utilisé pour la description est la simulation des différents exemples présentés dans cette thèse. Le but de cette présentation est de montrer les caractéristiques qu'un outil pour 42 devrait avoir. Dans un premier temps, nous présentons l'écriture des composants de base. Ensuite, nous verrons comment décrire des architectures en assemblant des composants pour définir de nouveaux composants. Enfin, nous présenterons le processus d'instantiation et de simulation de systèmes, ainsi qu'une interface graphique pour décrire des architectures.*

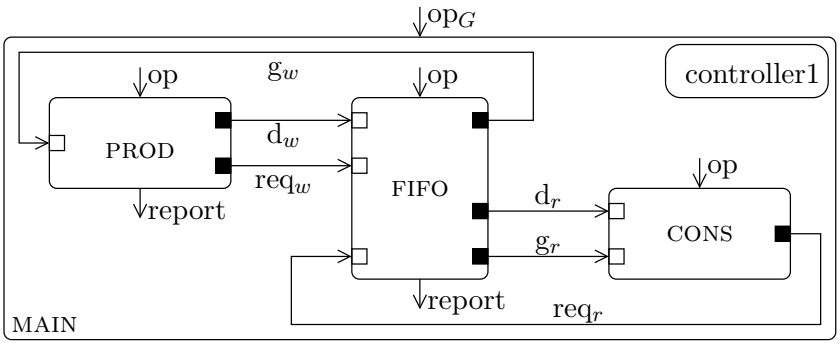


Fig. 9.1: The model of the producer consumer system

9.1 Writing Components and Architectures

In this section we are about to present how to design systems. We describe how components are designed, and how they are assembled to form new components by means of the example of Figure 9.1. The example is the model of the producer consumer system described in Section 4.2.3. The same example was used in Section 6.2.2 to introduce controllers acting as contract interpreters.

The tool maintains a repository of already created components that may be used during the design of a new system. Each component is associated with at least two files. A file describing its interface, and a file describing its implementation (how it is designed). In the current version of the tool, these description files, together with the contracts are written in XML. In this chapter we describe each of these files for each type of component (basic/composed), by means of a simple syntax avoiding XML markups.

9.1.1 Basic Components

Figure 9.2 is the implementation of the basic component `PROD` of Figure 9.1 in our prototype tool. The implementation of basic components has to extend the abstract class `BasicComponent42`. This class implements some methods that are required when a component is instantiated. In particular, the methods that are related with the initialization of the ports of the basic component.

The class implementing a component (Figure 9.2) includes the declaration of the set of its internal variables, its input/output data ports, and its control ports. The input control ports of the component are implemented as methods (e.g., `op()`).

When the component is activated with an input control port, it is usually required that the corresponding method executes only part if its code. In the example, this is achieved by using a `switch` statement. A more elegant manner would be the use of some `yielding primitives` that the component would call to terminate the execution step.

9.1.1.1 Interface Description

The classes implementing the behavior of basic components are compiled into *byte-code* classes that are instantiated when the component is used somewhere. To facilitate the use of such classes, a component is also equipped with a description of its interface. Figure 9.3 illustrates the interface description file of the component PROD of Figure 9.1. It consists in the description of the set of input/output data/control ports of the component; i.e., their name, their data type, etc.

```
public class PROD extends BasicComponent42 {
  // internal variables
  private int state;
  // port declaration
  private  ID <Boolean> g_w;
  private  OD <Integer> dw;
  ...

  // Constructors, initializing ports, etc.
  ...
  // Input control ports
  public void op(){
    switch(state){
      case 0:
        req_w.write(new Boolean(true));
        state=1;
        break;
      case 1:
        Boolean g = g_w.read();
        if(g.booleanValue()) state=2;
        else state = 0;
        break;
      case 2:
        dw.write(new Integer(42));
        state=0;
        break;
} } }
```

Fig. 9.2: The implementation of the component PROD in Java

```
interface PROD_ITF{
   port {name: op, data_type: bool, type: ic}
   port {name: g_w, data_type: bool, type: id}
   port {name: d_w, data_type: int, type: id}
   port {name: req_w, data_type: bool, type: od}
   port {name: report, data_type: {ok, ko}, type: oc}
}
```

Fig. 9.3: Interface description file of the component PROD

```
component PROD{
    interface: PROD_ITF
    contract: PROD-CT
    basic {javaclass: PROD}
    }
```

Fig. 9.4: Implementation description file of the component PROD

```
interface MAIN_ITF{
  port {name: opG, data_type: bool, type: ic}
}
```

Fig. 9.5: Interface description file of the component MAIN

9.1.1.2 Implementation Description

As we said previously a component should be associated with a file describing how it is designed. For instance, Figure 9.4 illustrates the implementation file associated with the component PROD. The implementation file associated with a basic component refers to the interface of the component, the Java class defining its behavior, and potentially to the contract associated with the component.

9.1.2 Composed Components

Composed components are designed as assemblies of existing components that are picked from the repository. The Component MAIN of Figure 9.1 is made of the three components. What we need to know about a composed component is its subcomponents, how they are connected, and which controller manges their execution. All of these information are in the implementation file.

9.1.2.1 Interface Description

Basic and composed components are not distinguishable from the outside. Hence, the description of the interface of a composed component does not differ from the interface description of a basic one. For the composed component MAIN, the interface description would only declare its input control port $\mathtt{op}_G$ (Figure 9.5).

9.1.2.2 Implementation Description

The implementation description file gathers all the information we need about the composed component. Figure 9.4 is the one associated with the component MAIN of Figure 9.1. It tells that the component MAIN is associated with the interface MAIN-ITF (of Figure 9.5), and the contract MAIN-CT. It also tells that MAIN is composed of three components (PROD, CONS, and FIFO). The architecture of the system is described by the set of wires. With each wire is associated a name, a source port and a sink port. Moreover, the implementation file of a composed component refers to the controller associated with the assembly.

9.1.3 Controllers

The controllers described so far in this thesis are of two types. Controllers defined by means of an imperative style language, and controllers defined as contract interpreters.

```
component MAIN{
  interface: MAIN_ITF
  contract: MAIN_CT
  composed{
    subcomponent{ name: PROD,  type: PROD}
    subcomponent{ name: CONS,  type: CONS}
    subcomponent{ name: FIFO,  type: FIFO}
    architecture{
      wire{ name: g_w, source: FIFO.g_w,  sink: PROD.g_w}
      wire{ name: d_w, source: PROD.d_w, sink: FIFO.d_w}
      wire{ name: req_w, source: PROD.req_w, sink: FIFO.req_w}
      ...
    }
    controller{ name: ctrl, type: language, file: ctrl.ctl}
  }
}
```

Fig. 9.6: Description file of the system in Figure 9.1

The element `controller` in the implementation file of a composed component (e.g., Figure 9.6) defines the name associated with the controller, its type, and potentially the path to a file containing the controller code (see below). In the attribute `type` of the element `controller` is allowed:

- **language** in case the controller is written in the imperative style language used so far. The attribute `file` has to be filled, it refers to the file containing the code of the controller.
- **interpreter** in case we use a contract interpreter that has accesses to the explicit control contracts of each component (see Section 6.2.2.2).
- **bottom-interpreter** in case we deal with hierarchic contract interpreters, where the concerned controller is at the bottom of the hierarchy.
- **top-interpreter** in case we deal with hierarchic contract interpreters, where the concerned controller is at the top of the hierarchy.
- **middle-interpreter** in case we deal with hierarchic contract interpreters, where the concerned controller is used in the intermediate levels of the hierarchy.

9.1.4 Contracts

In the same spirit of interface and implementation description, the contract of each component is described by means of an XML file. It describes the set of initial states together with the transitions of the contract. It also refers to the interface of the component it is associated with. Figure 9.7 illustrates part of the contract associated with the component PROD.

9.2 An Execution Engine to Perform Simulations

Once a component is designed, the tool allows for creating an instance of it. The instance of the component may be activated with its control inputs in order to performs simulations.

```
contract PROD_CT{
  initials{
    state{ name: p0}
  }
  transitions{
    transition{ lab: {}op{req_w}, src: p0, sink: p1}
    transition{ lab: {g_w=f}op/report{}, src: p1, sink: p0}
    ...
  }
}
```

Fig. 9.7: Part of the contract of the component PROD

```
Instantiate (component x)
  read implementation file
  read interface file
  create ports
  if(basic) then
     instantiate Java class
  endif
  if(composite) then
     for each subcomponent i do
       Instantiate(i)
     endfor
     connect components with wires
     instantiate the controller
  endif
```

Fig. 9.8: The instantiation process

9.2.1 Instantiation of Systems

The existing components are maintained in a repository from which the tool can access to the details of the components; i.e., their interface, implementation, Java classes, contracts, etc. In order to instantiate a system, the tool requires the reference to the component at the highest level of the hierarchy. It then follows the algorithm described in Figure 9.8. For example, `Intantiate(`MAIN`)` would create an instance of the system described in Figure 9.1)).

The instantiation algorithm is as fallows: first, it reads the implementation details and create the interface of the component. If the component is a basic one, it just creates an instance of the Java class implementing its behavior. If the component is a composed one, the tool then instantiates all its subcomponents, connect them and create an instance of the controller that defines how they should be activated.

When it comes to instantiate a controller, the tool instantiate a Java class that implements the behavior of the controller. Depending on the type of the controller, the Java class may be an interpreter of the imperative style language, or a contract interpreter.

The instantiation process parses the architecture of the components (the implementation description files) starting form the top level to the leafs of the hierarchy. It then instantiates components starting from those that are at the leafs, i.e., the basic ones.

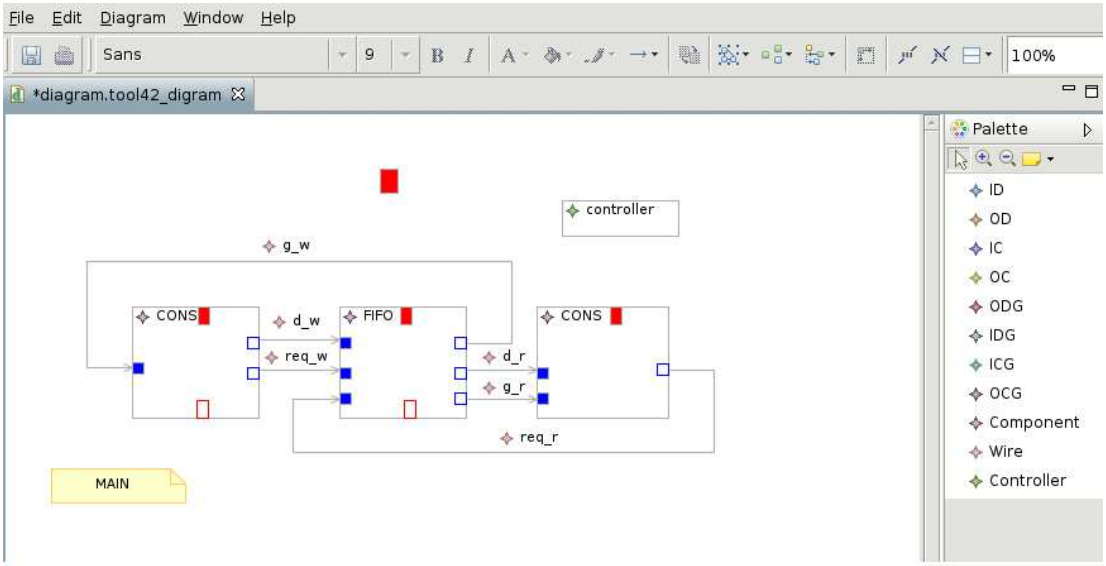


Fig. 9.9: A screen shot of the graphical interface of the tool

9.2.2 Simulation

The simulation process consists in creating an instance of a system, and activating it continuously with its control inputs. The activation of the component is nothing more than the call to the Java method that defines the behavior; it relies on the Java package *reflection*.

The simulation engine does not interferes with the behavior of the system. That is, the observed behavior is that exposed by the implementation of the controllers and the basic components.

9.2.3 A Graphical Interface

The tool provides a graphical editor for creating components and architectures in order to facilitate the design of systems. Figure 9.9 illustrates a screen shot of the editor. It allows to automatically generating the set of description files associated with the components, and a skeleton of the Java class implementing the behavior of a basic component.

The editor is developed within the GMF (Graphical Modeling Framework) plug-in of Eclipse, that provides generative component and runtime infrastructure for graphical editors. For the moment there is no direct connection between the simulation tool and the graphical editor.

Chapter 10

Conclusion & Prospects

10.1 Summary

The questions that motivated the 42 modeling approach are related to the design and the simulation of embedded systems. Facing the complexity of these systems and the difficulty to find an optimal solution, engineers are used to perform simulations. Various tools have shown their effectiveness in providing virtual prototypes of embedded systems long before the final system is available.

Because of time-to-market constraints, the embedded systems are mainly made of components. Hence the virtual prototypes should also be component-based. Most of the virtual prototyping tools favor a modular approach. However, they do not have a clear definition of components and do not provide a framework for reasoning on how these components may be assembled to form a system.

We recall the main challenges that motivated the design of the 42 approach:

- provide a language-independent component-based framework for modeling hardware software systems.
- provide support for a clean definition of the components, and help enforcing the FAMAPASAP (*Forget As Much As Possible As Soon As Possible*) principle.
- provide support for integration of existing modeling and simulation tools in open virtual prototyping environments.

10.1.1 Contributions

Components for Embedded Systems The major contribution of the work presented in this thesis lies in the complete definition of the 42 component-based modeling approach. We designed 42 with the idea of applying the FAMAPASAP principle. In order to enforce this principle, we decided to decouple the *control* from the *data* flows. This has the benefit of clearly describing component communications and how the control passes from a component to another. This has proven crucial for the application to SystemC/TLM.

The specification language in the form of control contracts is also a crucial point for the applicativity of the approach. Moreover, since contracts are executable, it helps obtaining lightweight simulation models.

A Rich Suite of Examples In order to be convinced by the expressiveness of 42, we developed a suite of modeling examples. The examples deal with the modeling of various *MoCCs*

ranging from pure synchrony to pure asynchrony as well as heterogeneous models. The results of these experiments helped in tailoring the basis of 42.

Virtual Prototyping of Hardware/Software Systems We experimented the usability of 42 as a tool for the virtual prototyping of hardware. The idea being to execute some embedded software on the virtual prototype, as it is done in the domain of systems-on-a-chip. In the context of this experiment we showed how the contracts may be executed in order to reason on component synchronizations early in the design flow. We also showed how the actual embedded software may be executed together with the contracts, which helps checking the compatibility of the software with its specification.

Using 42 Jointly with Other Approaches We provided a complete case-study on the use of 42 together with an existing component-based approach. We chose SystemC/TLM since it is the de-facto standard for the virtual prototyping of systems-on-a-chip. We demonstrated how expressive the 42 model is in describing TLM components. In particular, it insists on the interface TLM components should have, and how the contracts may describe their behavior. Moreover, we presented the possibility of generating 42 components from SystemC/TLM components, as well as the possibility of executing 42 and SystemC/TLM components with the SystemC scheduler or the 42 controllers. This shows the interoperability of the two approaches.

The 42-ization of SystemC/TLM may considered as a first step towards a framework providing support for the integration of existing modeling approaches and simulation tools in open virtual prototyping environments. The framework would consist of a set of tools for describing models from distinct approaches in the same formalism (e.g., 42-ization) and to execute them together.

A Toolset for Executing 42 models For the purpose of experiments we implemented a tool to allow for the design of 42 components and assemblies as well as the simulation of these systems. The tool takes into account the execution of 42 components given as detailed implementations (in languages like C, Java, Lustre, etc.), as well as the execution of control contracts. Moreover we are capable of importing components from other approaches and executing them together with 42 ones.

10.2 Prospects

42 is a new component-based approach to the modeling of heterogeneous embedded systems. Its actual version is expressive enough to model various *MoCCs*. The model may be extended in order to gain more expressiveness, or to cover other modeling aspects. Future research directions are numerous, they may include the points listed in the sequel.

10.2.1 Semantical Aspects

Along with the examples described in this thesis, we showed that the controllers are expressive enough to describe various *MoCCs*. When it comes to model parallel activities of components, we rely on a non-deterministic controller that produces interleaving of components' activations. The question of whether there would be a need for expressing parallelism in the controllers has never been examined, because simulation models do not need such a feature. But this could be investigated.

Another direction of research would be related to the notion of time in the 42 model. Indeed, in almost all the examples we described, we dealt with a logical notion of time, except in the

synchronous modeling of programs where we inherit a notion of quantitative time (counting clock ticks). Logical time is adopted by several modeling approaches and has proven useful. However, it does not give solutions to all problems related to timing aspects. For that purpose, we could investigate how to deal with general quantitative time in 42.

10.2.2 The Language of Control Contracts

Another direction of further work would be related to the language used for control contracts. This includes enhancement of their *readability* and *expressiveness*.

Readability: The examples of contracts we have seen so far are small enough to be understandable. However, a contract for a component exposing a complex behavior may be a huge automaton. The Greek variables used to refer to the control outputs of a component play an important role for enhancing the readability of the control contracts. We can just compare the original form of a contract with its expanded version to see the difference (see Section 5.2.2). In the same sense, we can introduce more variables from finite-domains to keep a small set of states (e.g., counters).

Sometimes the behavior of a component may be expressed in terms of a simple combination of the behaviors of its subcomponents. In this situation, it is useful to express the contract of a component in terms of simple operations on the set of its subcomponent contracts. By simple operations we mean for instance, synchronous or asynchronous product of automata, asynchronous product with synchronization, etc.

Expressiveness: In the simulation and execution of control contracts, we are not limited in the expressiveness of the language of contracts as it would be the case if we were interested in verification. For that purpose, we could think about a more expressive language than automata for writing contracts. This would deserve some work, in particular, in the definition of the required expressiveness of that language (e.g., non-determinism, variables with infinite domains, etc.).

10.2.3 Towards Non-Functional Properties

An embedded system must satisfy non-functional properties as well as functional ones. In the context of this thesis we mainly deal with the functional aspects of an embedded system. However, thinking about non-functional properties, typically *energy consumption*, is becoming unavoidable. This is true for embedded systems in general and for sensor networks or consumer electronics (mobile phones and all kinds of portable devices) in particular, because of lifetime or autonomy constraints.

There is a need for high-level models to reason about non-functional aspects for embedded systems. They would allow for taking design decisions satisfying functional as well as non-functional properties, early in the design cycle. Therefore, there is a need for describing functional and non-functional models, as well as their relation with each other. This challenging problem initiated the HELP (*High Level Models for Low Power Systems*) project [1].

We did some preliminary experiments (see [BMAM08] below) with 42 in order to answer some questions related to functional and non-functional modeling of embedded systems. These experiments raised the following points:

[1] an ANR Arpège project http://www-verimag.imag.fr/PROJECTS/SYNCHRONE/HELP/

- The interface between the functional and the non-functional parts of a component: e.g., the influence of the functionality on the consumption in one component.
- The non-functional interface between components: e.g., how to deduce the consumption of an assembly from the consumption of its components.
- The organization of functional and non-functional models: as a hierarchy of functional-components next to a hierarchy of non-functional ones, or as a hierarchy of mixed functional/non-functional components.

Thanks to its interoperability with SystemC/TLM, 42 is a good candidate for the HELP project. SystemC/TLM would be used to design virtual prototypes modeling the functional aspect of embedded systems. 42 would then be used as framework for integrating these functional models with non-functional ones from other approaches.

10.3 Publications Related to 42

Conference Papers

BM09 Tayeb Bouhadiba and Florence Maraninchi. Contract-based coordination of hardware components for the development of embedded software. In John Field and Vasco Thudichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2009.

BMF09 Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal. Formal and executable contracts for transaction-level modeling in systemc. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *EMSOFT*, pages 97–106. ACM, 2009.

MB07 Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 53–62. ACM, 2007.

Workshops

MB07+ Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In SYNCHRON'07 International Open Workshop on Synchronous Programming.

BMAM08 Tayeb Bouhadiba, Florence Maraninchi, Karine Altisen, Matthieu Moy. Computational Modeling of Non-Functional Properties with the Component Model 42. Position paper for the ARTIST MoCC'08 Workshop, july 2008, Eindhoven

Chapter 10'

Conclusion & Perspectives (In French)

10'.1 Résumé

Les questions qui ont motivé la définition de l'approche 42 sont liées à la modélisation et à la simulation des systèmes embarqués hétérogènes. Pour faire face à la complexité de ces systèmes, et à la difficulté à trouver une solution optimale, les approches de développement adoptées par les ingénieurs reposent sur la simulation. Ainsi, un large éventail d'outils est disponible pour fournir des modèles de simulation et prototypes virtuels bien avant que le système réel ne soit disponible.

Les contraintes liées au temps de mise sur le marché poussent les développeurs de systèmes embarqués à adopter des approches de conceptions par assemblage de composants. Cela est particulièrement claire pour le développement de la plateforme matérielle. Ainsi, les approches de prototypage virtuel sont intrinsèquement composants. Cependant, la plupart des outils de prototypage virtuel adoptent une approche modulaire pour le développement de prototypes virtuels, mais n'ont pas une définition claire de la notion de composants. En plus, ils ne fournissent pas d'environnement adéquat pour réfléchir sur les composants et comment ils peuvent être assemblés pour former un système.

Nous rappelons les challenges qui ont motivé la définition du modèle 42:

- fournir un environnement, indépendant de tout langage ou formalisme, pour la modélisation par composants de systèmes matériels/logiciels.
- fournir un support pour une définition claire de la notion de composants, et aider à appliquer le FAMAPSAP.
- fournir un support pour l'intégration de modèles existants, issus d'outils hétérogènes, dans un environnement de prototypage virtuel ouvert.

10'.1.1 Contributions

Composants pour les Systèmes Embarqués La contribution majeure des travaux de cette thèse consiste en la définition de l'approche à composants 42. 42 a été défini dans le but de fournir un environnement de prototypage virtuel des systèmes embarqués qui aide à appliquer le principe du FAMAPSAP. A cet effet, une première étape a été de dissocier le flot de contrôle du flot de données. L'avantage de cette séparation est de rendre explicite la communication entre les composants, et le passage du flot de contrôle d'un composant à un autre. Cette séparation

a favorisé la description de plusieurs *MoCCs*, et s'est avérée très intéressante pour l'application de 42 à SystemC/TLM.

Le langage de spécification sous la forme de contrats de contrôle a été un point crucial dans l'application de 42 à SystemC/TLM. De plus, comme les contrats de contrôle sont exécutables, il a été possible d'obtenir des modèles de simulation légers, qui nous permettent d'observer la synchronisation entre les composants TLM.

Un Riche Ensemble d'Exemples de Modélisation Afin d'être convaincu de l'expressivité du modèle 42, nous avons développé une série d'exemples de modélisation. Les exemples que nous avons présenté consistent en la modélisation de plusieurs *MoCCs* allant du pure synchrone au pure asynchrone, ainsi que des *MoCCs* hétérogènes. Les résultats des expériences de modélisation ont permit d'affiner les éléments de base de 42.

Prototypage Virtuel de Systèmes Matériel/Logiciel Dans le contexte du prototypage virtuel des systèmes-sur-puce, nous avons effectué des expérimentations pour étudier l'utilisabilité de 42 en tant qu'approche pour le développement de prototypes virtuels du matériel. Le but étant de fournir un support sur lequel le logiciel embarqué pourra être exécuté. L'utilisation des contrats de contrôle en tant qu'abstractions non-déterministes du comportement des composants nous a fourni un support pour réfléchir sur la synchronisation des composants, très tôt, avant que l'implémentation des composants ne soit détaillée. Nous avons développé un mécanisme d'exécution de contrats et d'implémentations ensemble. Nous avons étendu ce mécanisme d'exécution afin de pouvoir exécuter le logiciel embarqué sur le prototype virtuel du matériel. Ce mécanisme nous permet aussi de tester la compatibilité des implémentations (ainsi que le logiciel embarqué) avec les leurs contrats.

L'utilisabilité de 42 avec des Approches Existantes Dans le but de montrer l'utilisation conjointe de 42 avec des approches existantes, nous avons fourni un cas d'étude complet de modélisation de SystemC/TLM en 42. Nous avons choisi SystemC/TLM vu qu'il est l'un des standards dans l'industrie pour le prototypage virtuel des systèmes-sur-puce. Nous avons montré que l'expressivité du modèle 42 est suffisante pour décrire des composants SystemC/TLM. SystemC/TLM étant assez complexe, sa modélisation en 42 montre les détails cachés par la mécanique de simulation de SystemC. En particulier, nous avons pu décrire l'interface que doit avoir un composant TLM, indépendamment de la mécanique de simulation de SystemC.

Dans ce cas d'étude, nous avons montré aussi, l'utilisation des contrats de contrôle pour décrire les comportements des composants TLM et comment les exploiter pour fournir des modèles de simulation légers, utilisables comme support pour la réflexion sur la synchronisation des composants TLM. De plus, nous avons présenté la possibilité d'extraire les contrats de contrôle depuis du code SystemC, ainsi que la possibilité d'exécuter des composants 42 et SystemC/TLM avec l'ordonnanceur SystemC ou bien avec les contrôleurs 42. Ceci montre l'interopérabilité des deux approches.

La 42-isation de SystemC/TLM peut être considéré comme un premier pas vers un environnement de prototypage ouvert, qui permet l'intégration de différentes approches de simulation et de modélisation. Cet environnement consisterait en un ensemble d'outils pour la description de modèles provenant de différentes approches dans le même formalisme (e.g., la 42-isation) et les exécuter ensemble.

Un Outil pour Simuler des Modèle écrits en 42 Afin de pouvoir simuler et exécuter des modèles écrits en 42, nous avons développé un outil qui nous permet d'écrire des composants 42, de les assembler, et de les simuler. L'outil permet l'exécution de composants décrits par

des implémentations concrètes (écrite en un langage tel que Java, C, Lustre, etc.), ainsi que l'exécution des contrats de contrôle. De plus, nous somme capable d'importer des composants existants, provenant d'autres approches, et de les exécuter dans la mécanique de simulation de 42.

10'.2 Perspectives

42 est une approche à composants pour le prototypage virtuel des systèmes embarqués hétérogènes. Sa version actuelle, telle qu'elle est décrite dans cette thèse, est assez expressive pour décrire différents *MoCCs*. Le modèle peut être étendu pour augmenter son expressivité, ou pour couvrir d'autres aspects de modélisation. Plusieurs directions sont possible pour les travaux futurs ; ils pourraient inclure les points suivants :

10'.2.1 Aspect Sémantiques

Durant la description des exemples de modélisation, nous avons montré que les contrôleurs sont assez expressifs pour décrire différents *MoCCs*. Quand il y a eu besoin de modéliser des activités parallèles, nous avons utilisé des contrôleurs non-déterministes qui produisent des *interleavings* d'activation de composants. Jusqu'à maintenant, nous n'avons pas regardé si l'expression du parallélisme dans les contrôleurs est une propriété indispensable. La question de l'expression du parallélisme dans les contrôleurs 42 est une piste de recherche futurs.

Une autre direction de recherche est liée à la notion du temps dans les modèles 42. En effet, dans la plupart des exemples que nous avons donné, nous nous somme contenté d'une notion logique du temps, à part pour la modélisation du synchrone où nous héritons d'une notion de temps quantitatif (les ticks d'horloge). Le temps logique est adopté par plusieurs modèles de simulation et a été prouvé utile. Cependant, il ne fourni pas de solutions pour toutes les questions liées au temps. Pour 42, un des travaux futurs serait de réfléchir à comment gérer une notion quantitative du temps, d'une manière générale.

10'.2.2 Le Langage des Contrats de Contrôle

Les autres directions des travaux futurs concernent le langage utilisé pour écrire des contrats de contrôle. Ceci implique l'augmentation de leur *lisibilité* et leur *expressivité*

Lisibilité: Les exemples de contrats que nous avons vu jusqu'à maintenant sont assez petits pour être compréhensibles. Cependant, un contrat pour un composant au comportement complexe peut être décrit par des automates assez gros. Les lettres Grecs que nous avons utilisé pour faire référence aux sorties de contrôle d'un composant jouent un rôle important dans la lisibilité d'un contrat. La comparaison entre la version originale d'un contrat et sa version expansée nous montre l'apport de ces variables (voir la section 5.2.2). Dans le même esprit, sans rajouter à l'expressivité des contrats, nous pouvons introduire d'autres variables d'un domaine fini pour garder un petit ensemble d'états (e.g., compteurs).

Parfois, le comportement d'un composant composite peut être décrit par une simple combinaison des comportements de ses sous-composants. Dans cette situation, il peut être intéressant d'exprimer le contrat du composant composite en terme d'opérations simples sur les contrats de ses sous-composants. Par exemple, nous pouvons décrire le contrat d'un composants comme étant le produit synchrone, asynchrone, etc. des contrats de ces sous-composants.

Expressivité: L'objectif de 42 est d'identifier les éléments de base nécessaires à la description des *MoCCs*. Nous ne cherchons pas à limiter son expressivité pour rentrer, par exemple, dans un contexte de vérification formelle. Cette remarque est aussi valable pour les contrats de contrôle. On peut penser par exemple à un langage plus expressif que de simples automates pour écrire les contrats. Identifier un nouveau langage pour les contrats de contrôle mérite quelques travaux de recherche, essentiellement pour déterminer l'expressivité requise (e.g., non-determinisme, variables d'un domaine infini, etc.).

10'.2.3 Vers la Modélisation de Propriétés Non-Fonctionnelles

En plus des propriétés fonctionnelles, les systèmes embarqués doivent souvent satisfaire des propriétés non-fonctionnelles. Dans cette thèse, nous nous somme focalisé sur la modélisation de l'aspect fonctionnel d'un système. Cependant, modéliser les propriétés non-fonctionnelles, typiquement la *consommation en énergie*, est indispensable. Ceci est vrai pour les systèmes embarqués en général, et pour les réseaux de capteurs ou l'électronique grand public (e.g., téléphones mobiles et équipements portables) en particulier, à cause des contraintes liées à leur autonomie ou leur durée de vie.

Pour la modélisation des systèmes embarqués, il y a besoin de modèles haut-niveaux pour réfléchir sur leurs aspects fonctionnels et non-fonctionnels. Ces modèles permettraient de prendre des décisions sur la conception d'un système, très tôt dans le flot de conception, afin de satisfaire à la fois les contraintes fonctionnelles et non-fonctionnelles. L'influence du fonctionnel sur le non-fonctionnel (et inversement) étant de plus en plus significative, il y a besoin de formalismes capables de décrire des aspects fonctionnels et non-fonctionnel ainsi que la relation qui existe entre eux. Ce problème de modélisation a motivé l'initiation du projet HELP (*High Level Models for Low Power Systems*) [1].

Nous avons effectué quelques expérimentations (voir [BMAM08] ci-dessous) avec 42 afin de répondre à quelques questions liées à la modélisation des aspects fonctionnels et non-fonctionnels des systèmes embarqués. Ces expérimentations ont soulevé les points suivantes:

- Quelle est l'interface entre les parties fonctionnelles et non-fonctionnelles d'un composant : e.g., l'influence de la fonctionnalité sur la consommation en énergie d'un composant.
- Comment organiser des modèles fonctionnels et non-fonctionnels : comme une hiérarchie de composants fonctionnels en parallèle à une hiérarchie de composants non-fonctionnels, ou bien comme une seule hiérarchie où les composants fonctionnels et non-fonctionnels sont mélangés à tous les niveaux.

Grâce à son interopérabilité avec SystemC/TLM, 42 est un bon candidat pour le projet HELP. Dans ce projet, SystemC/TLM serait utilisé pour la conception de prototypes virtuels, modélisant ainsi des propriétés fonctionnelles d'un système. 42 serait utilisé comme environnement pour l'intégration des ces prototypes fonctionnels avec des modèles non-fonctionnels issus d'autres approches.

10'.3 Publications Autour de 42

Publication dans des Conférences Internationales

BM09 Tayeb Bouhadiba and Florence Maraninchi. Contract-based coordination of hardware components for the development of embedded software. In John Field and Vasco Thu-

[1] Un projet ANR Arpège http://www-verimag.imag.fr/PROJECTS/SYNCHRONE/HELP/

dichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2009.

BMF09 Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal. Formal and executable contracts for transaction-level modeling in systemc. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *EMSOFT*, pages 97–106. ACM, 2009.

MB07 Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 53–62. ACM, 2007.

Workshops

MB07+ Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In SYNCHRON'07 International Open Workshop on Synchronous Programming.

BMAM08 Tayeb Bouhadiba, Florence Maraninchi, Karine Altisen, Matthieu Moy. Computational Modeling of Non-Functional Properties with the Component Model 42. Position paper for the ARTIST MoCC'08 Workshop, july 2008, Eindhoven

Bibliography

[AAAG+05] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, and Tony Tseng. Modeling and implementing software architecture with acme and archjava. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 676–677, New York, NY, USA, 2005. ACM. 148

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986. 152

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. 148

[ALM10] Karine Altisen, Yanhong Liu, and Matthieu Moy. Performance evaluation of components using a granularity-based interface between real-time calculus and timed automata. In *Eighth Workshop on Quantitative Aspects of Programming Languages (QAPL)*, Paphos, Cyprus, March 2010. 21

[AM10] Karine Altisen and Matthieu Moy. Arrival curves for real-time calculus: the causality problem and its solutions. In J. Esparza and R. Majumdar, editors, *TACAS*, pages 358–372, March 2010. 21

[And04] Charles André. Computing syncharts reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, October 2004. http://www.sciencedirect.com; doi:10.1016/j.entcs.2003.05.007. 22

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004. 152

[Ban96] Mario Banville. Sonia: An adaptation of linda for coordination of activities in organisations. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 57–74, London, UK, 1996. Springer-Verlag. 152

[BBS98] Daniel Brand, Reinaldo A. Bergamaschi, and Leon Stok. Don't cares in synthesis: theoretical pitfalls and practical solutions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(4):285–304, 1998. 155

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society. 148

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press. 37

[BCCSV05] Albert Benveniste, Benoît Caillaud, Luca P. Carloni, and Alberto Sangiovanni-Vincentelli. Tag machines. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 255–263, New York, NY, USA, 2005. ACM Press. 146

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003. 20

[BCL+06a] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in java. *Softw, Pract. Exper*, 36(11-12):1257–1284, 2006. 33, 148, 150, 153

[BCL+06b] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006. 150

[Ber92] Gerrard Berry. A hardware implementation of pure esterel. *Sadhana*, 17(1):95–130, 1992. 27

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. 24, 147

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999. 33

[BL02] Dag Björklund and Johan Lilius. A language for multiple models of computation. In *CODES '02: Proceedings of the tenth international symposium on Hardware/-software codesign*, pages 25–30, New York, NY, USA, 2002. ACM. 147

[BLG90] Albert Benveniste and Paul Le Guernic. Hybrid dynamical systems theory and the signal language. In *IEEE Transactions on Automatic Control*, pages 535–546. IEE, 1990. 80

[BM09] Tayeb Bouhadiba and Florence Maraninchi. Contract-based coordination of hardware components for the development of embedded software. In John Field and Vasco Thudichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2009. 11, 17

[BMF09] Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal. Formal and executable contracts for transaction-level modeling in systemc. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *EMSOFT*, pages 97–106. ACM, 2009. 11, 17

[Bou07] Yussef Bouzouzou. Accélération des simulations de systèmes sur puce au niveau transactionnel. Diplôme de recherche technologique, Université Joseph Fourier, 2007. 29

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM. 37

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006. 152

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003. 28

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976. 37

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 37

[CCG+04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Giuseppe Maruccia, and Francesco Papariello. Occn: A network-on-chip modeling and simulation framework. In *DATE*, pages 174–179. IEEE Computer Society, 2004. 10, 16

[CCM] Object Management Group: CORBA Components, v. 3.0, OMG document formal/02-06-65. 33

[CDE+05] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. Synchronization of periodic clocks. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 339–342, New York, NY, USA, 2005. ACM. 80, 83

[CDE+06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. -synchronous kahn networks: a relaxed model of synchrony for real-time systems. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL Symposium on Principles of Programming Languages*, pages 180–193. ACM, 2006. 80, 83

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. 35

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989. 152

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999. 35, 36

[CGS94] Thomas Cheatham, Haiming Gao, and Dan C. Stefanescu. A suite of analysis tools based on a general purpose abstract interpreter. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 188–202, London, UK, 1994. Springer-Verlag. 37

[CMMC08] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 9–14, New York, NY, USA, 2008. ACM. 67

[CMP01] Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. *Lecture Notes in Computer Science*, 2187:215–226, 2001. 21

[Con] The Spirit Consortium. IP-XACT 1.4 specification. `www.spiritconsortium.org/releases/1.4`. 138

[Cor08] Jérôme Cornet. *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*. PhD thesis, Institut National Polytechnique de Grenoble, 2008. 29

[Cou97] Patrick Cousot. Types as abstract interpretations. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331, New York, NY, USA, 1997. ACM. 37

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM. 79

[CR05] Feng Chen and Grigore Rosu. Java-mop: A monitoring oriented programming environment for java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005. 36

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *Proceedings of the Ninth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 26(5):109–120, 2001. 34, 153

[Dev91] Srinivas Devadas. Optimizing interacting finite state machines using sequential don't cares. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 10(12):1473–1484, 1991. 155

[DM93] Maurizio Damiani and Giovanni De Micheli. Don't care set specifications in combinational and synchronous logic circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(3):365–388, 1993. 155

[DMN90] Srinivas Devadas, Hi-Keung Tony Ma, and A. Richard Newton. Redundancies and don't cares in sequential logic synthesis. *J. Electronic Testing*, 1(1):15–30, 1990. 155

[EJB03] Enterprise Java Beans specifcation, version 2.1. Sun Microsystems, November 2003. 33

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. 10, 16, 144

[Fal09] Ylies C. Falcone. *Étude et mise en oeuvre de techniques de validation à l'exécution*. Thèse de doctorat, Université Joseph Fourier, Grenoble, November 2009. 36

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2009. ACM. 36

[FLVC05] Peter H. Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. *An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering*, volume 176/2005 of *IFIP International Federation for Information Processing*, pages 3 – 15. Springer Boston, 2005. 28

[FS02] Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 995–1012, London, UK, 2002. Springer-Verlag. 153

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think: A software framework for component-based operating system kernels. In Carla Schlatter Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 73–86. USENIX, 2002. 150

[GBA+09] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, 2009. 147

[GG87] Thierry Gautier and Paul Le Guernic. Signal: A declarative language for synchronous programming of real-time systems. In *FPCA*, pages 257–277, 1987. 24, 147

[GG03] Abdoulaye Gamati and Thierry Gautier. Synchronous modeling of avionics applications using the signal language. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:144, 2003. 21

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006. 24

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 10, 11, 16, 17, 28, 148

[Gir94] Alain Girault. *Sur la Répartition de Programmes Synchrones*. Phd thesis, INPG, Grenoble, France, January 1994. 24

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997. 125

[Goe98] M. Goel. Process networks in ptolemy ii. Technical Report UCB/ERL M98/69, EECS Department, University of California, Berkeley, 1998. 145

[Gon07] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, October 2007. 24

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Springer, 1 edition, March 2000. 28

[Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. 20

[HB02] Nicolas Halbwachs and Siwar Baghdadi. Synchronous modelling of asynchronous systems. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 240–251, London, UK, 2002. Springer-Verlag. 147

[HB08] Cécile Hardebolle and Frédéric Boulanger. Modhel'x: A component-oriented approach to multi-formalism modeling. pages 247–258, 2008. 147

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 144

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 24, 80, 147

[HFG08] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS '08*, pages 131–136, New York, NY, USA, 2008. ACM. 142

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992. 20

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag. 35

[HM06] Nicolas Halbwachs and Louis Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society. 21

[HMMCM06] C. Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In *Formal Methods in Computer-Aided Design*, pages 171–178. IEEE Computer Society, 2006. 125

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 153

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 37, 142

[HP96] Gerard J. Holzmann and Doron Peled. The state of spin. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 385–389, London, UK, 1996. Springer-Verlag. 37

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI'08: 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 339–348. ACM, June 2008. 37

[Jan03] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 20

[JHR+07] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, and David Lesens. Virtual Execution of AADL Models via a Translation into Synchronous Programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software EMSOFT 2007*, pages 134 – 143, Salzburg Austria, 2007. ASSERT. 21

[JPSN09] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 110–120, New York, NY, USA, 2009. ACM. 36

[Kah62] Albert B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, November 1962. 62

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *ifip congress 74*, pages 471–475, 1974. 70, 71, 145

[KB77] Gilles Kahn and David B.MacQueen. Coroutines and networks of parallel processes. In *ifip congress 77*, pages 993–998, 1977. 70, 145

[Kra98] Reto Kramer. icontract - the java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society. 34

[Lar90] Kim Guldstrand Larsen. Modal specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 232–246, London, UK, 1990. Springer-Verlag. 154

[LAS00] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag. 37

[Lee99] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Ann. Softw. Eng.*, 7(1-4):25–45, 1999. 145

[LFLL06] Samper Ludovic, Maraninchi Florence, Mounier Laurent, and Mandel Louis. Glonemo: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense: First International Conference on Integrated Internet Ad hoc and Sensor Networks*, Nice, France, May 2006. IEEE. 10, 16, 21

[Liu98] Jie Liu. Continuous time and mixed-signal simulation in ptolemy ii. Technical Report UCB/ERL M98/74, EECS Department, University of California, Berkeley, 1998. 145

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. 142

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987. 154

[LZ05] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer, 2005. 147

[LZ07] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM. 144

[MA07] Sun Meng and Farhad Arbab. Web services choreography and orchestration in reo and constraint automata. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 346–353, New York, NY, USA, 2007. ACM. 152

[MB07] Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 53–62. ACM, 2007. 11, 17

[Mcm92a] Kenneth L. Mcmillan. The SMV system, November 06 1992. 142

[McM92b] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. 37

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: a constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304–312, 1994. 152

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. 34

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. 34

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 153

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983. 21

[MMC+08] Florence Maraninchi, Matthieu Moy, Jérôme Cornet, Laurent Maillet Contoz, Claude Helmstetter, and Claus Traulsen. SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation. In IEEE, editor, *2008 Joint IEEE-NEWCAS and TAISA Conference 2008 Joint IEEE-NEWCAS and TAISA Conference*, page unknown, Montréal Canada, 06 2008. B.6.3, D.2.4, D.3.1, F.4.3, F.3.1, B.8.1. 142

[MMMC05a] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *ACSD*, pages 26–35. IEEE Computer Society, 2005. 142

[MMMC05b] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT*, pages 317–324, September 2005. 135

[MR01] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Comput. Lang.*, 27(1/3):61–92, 2001. 22

[NH06] Bernhard Niemann and Christian Haubelt. Formalizing tlm with communicating state machines. In *FDL*, pages 285–293. ECSI, 2006. 142

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998. 152

[PBJ98] F. Plásil, D. Bálek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society. 33

[plo] Polyspace. http://www.mathworks.com/products/polyspace/. 37

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. 35

[PR09] Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 215–224, New York, NY, USA, 2009. ACM. 63

[PS08] Olivier Ponsini and Wendelin Serwe. A schedulerless semantics of tlm models written in systemc via translation into lotos. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. 142

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002. 34, 153

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag. 36

[Ray88] Pascal Raymon. *Compilation stparée de programmes LUSTRE*. Technical report, SPECTRE L5, IMAG, Grenoble, June 1988. 63

[RBB+09] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces: unifying interface automata and modal specifications. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 87–96, New York, NY, USA, 2009. ACM. 155

[RC03] Arnab Ray and Rance Cleaveland. Architectural interaction diagrams: Aids for system modeling. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 396–406, Washington, DC, USA, 2003. IEEE Computer Society. 147

[RH92] Frédéric Rocheteau and Nicolas Halbwachs. Implementing reactive programs on circuits: A hardware implementation of lustre. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 195–208, London, UK, 1992. Springer-Verlag. 27

[RHR91] Christophe Ratel, Nicolas Halbwachs, and Pascal Raymond. Programming and verifying critical systems by means of the synchronous data-flow language lustre. In *SIGSOFT '91: Proceedings of the conference on Software for citical systems*, pages 112–119, New York, NY, USA, 1991. ACM. 24

[Rin00] Thomas Ringler. Static worst-case execution time analysis of synchronous programs. In *Ada-Europe '00: Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, pages 56–68, London, UK, 2000. Springer-Verlag. 21

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS*, 15(4):391–411, 1997. 36

[SDK+95a] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995. 148, 152

[SDK+95b] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions in Software Engeneering*, 21(4):314–335, 1995. 148

[SPED06] Lionel Seinturier, Nicolas Pessemier, Clément Escoffier, and Didier Donsez. Towards a reference model for implementing the fractal specifications for java and the .net platform. In *Fractal CBSE workshop at ECOOP 2006, Nantes, July 3 2006, to appear at Springer Verlag LNCS Serie*, 2006. 150

[SPI] Spice. `bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/`. 147

[sysa] Synopsys, inc. cocentric(r) systemc compiler behavioral modeling guide, 2002. 27

[SYSb] System verilog. `www.systemverilog.org`. 28

[sys06] Ieee standard system c language reference manual. *IEEE Std 1666-2005*, pages 01–423, 2006. 28, 29

[TBO05] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. Qinna, a component-based qos architecture. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2005. 34

[Tho75] Jean U. Thoma. *Introduction to Bond-Graphs and their applications*. Pergamon Press, 1975. 147

[Tol96] Robert Tolksdorf. Coordinating services in open distributed systems with laura. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 386–402, London, UK, 1996. Springer-Verlag. 152

[VDBL89] Jan. Van Den Bos and Chris. Laffra. Procol: a parallel object language with protocols. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 95–102, New York, NY, USA, 1989. ACM. 34

[ver] Verilog. http://www.verilog.com. 27

[vhd92] Ieee standards interpretations: Ieee std 1076-1987, ieee standard vhdl language reference manual. *IEEE Std 1076/INT-1991*, page 1, 1992. 27

[vhd99] Ieee standard vhdl analog and mixed-signal extensions. *IEEE Std 1076.1-1999*, page i, 1999. 148

[VPB+08] Michel Vasilevski, Francois Pecheux, Nicolas Beilleau, Hassan Aboushady, and Karsten Einwich. Modeling refining heterogeneous systems with systemc-ams: application to wsn. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 134–139, New York, NY, USA, 2008. ACM. 148

[VVR06] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inf.*, 73(4):583–598, 2006. 34, 153

[Wei08] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. 28

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 34

[Zho05] G. Zhou. Dynamic data flow modeling in ptolemy ii. Technical Report UCB/ERL M05/7, EECS Department, University of California, Berkeley, Jan 2005. 145

Abstract

The work presented in this thesis deals with virtual prototyping of heterogeneous embedded systems. The complexity of these systems make it difficult to find an optimal solution. Hence, engineers usually make simulations that require virtual prototyping of the system. Virtual prototyping of an embedded system aims at providing an executable model of it, in order to study its functional as well as its non-functional aspects. Our contribution is the definition of a new component-based approach for the virtual prototyping of embedded systems, called 42. 42 is not a new language for the design of embedded systems, it is a tool for describing components and assemblies for embedded systems at the system-level.

Virtual prototyping of embedded systems must take into account their heterogeneous aspect. Following Ptolemy, several approaches propose a catalogue of MoCCs (Models of Computation and Communication) and a framework for hierarchically combining them in order to model heterogeneity. As in Ptolemy, 42 allows to organize components and MoCCs in hierarchy. However, the MoCCs in 42 are described by means of programs manipulating a small set of basic primitives to activate components and to manage their communication.

A component-based approach like 42 requires a formalism for specifying components. 42 proposes several means for specifying components. We will present these means an give particular interest to 42 control contracts.

42 is designed independently from any language or formalism and may be used jointly with the existing approaches. We provide a proof of concept to demonstrate the interest of using 42 and its control contracts with the existing approaches.

Keywords. Heterogeneous Embedded Systems, Components, Virtual Prototyping, Contracts and Specifications, MoCCs (Models of Computation and Communication), Synchronous/Asynchronous, Semantics.

Résumé

Les travaux présentés dans cette thèse portent sur le prototypage virtuel des systèmes embarqués hétérogènes. La complexité des systèmes embarqués fait qu'il est difficile de trouver une solution optimale. Ainsi, les approches adoptées par les ingénieurs reposent sur la simulation qui requiert le prototypage virtuel. L'intérêt du prototypage virtuel est de fournir des modèles exécutables de systèmes embarqués afin de les étudier du point de vue fonctionnel et non-fonctionnel. Notre contribution consiste en la définition d'une nouvelle approche à composants pour le prototypage virtuel des systèmes embarqués, appelé 42. 42 n'est pas un nouveau langage pour le développement des systèmes embarqués, mais plutôt un outil pour la description et l'assemblage de composants pour les systèmes embarqués, au niveau système.

Un modèle pour le prototypage virtuel des systèmes embarqués doit prendre en compte leur hétérogénéité. Des approches comme Ptolemy proposent un catalogue de MoCCs (Models of Computation and Communication) qui peuvent être organisés en hiérarchie afin de modéliser l'hétérogénéité. 42 s'inspire de Ptolemy dans l'organisation hiérarchique de composants et de MoCCs. Cependant, les MoCCs dans 42 ne sont pas fournis sous forme de catalogue, ils sont décrits par des programmes qui manipulent un petit ensemble de primitives de base pour activer les composants et gérer les communications entre eux.

Une approche à composants comme 42 requiert un formalisme de spécification de composants. Nous étudierons les moyens proposés par 42 pour décrire les composants. Nous nous intéresseront particulièrement aux contrats de contrôle de 42.

42 est indépendant de tout langage ou formalisme. Il est conçu dans l'optique d'être utilisé conjointement avec les approches existantes. Nous donnerons une preuve de concept afin de montrer l'intérêt d'utiliser 42 et les contrats de contrôle associés aux composants, conjointement avec des approches existantes.

Mots Clés. Systèmes Embarqués Hétérogènes, Composants, Prototypage Virtuel, Contrats et Spécifications, MoCCs (Models of Computation and Communication), Synchrone/Asynchrone, Sémantique.