Geometric semantics for asynchronous computability

Jérémy Ledent

NNT : 2019SACLX099

Sémantiques Géométriques pour la Calculabilité Asynchrone

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'École Polytechnique

École doctorale n°580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 12 décembre 2019, par

Jérémy Ledent

Composition du Jury :

Pierre Fraigniaud Directeur de Recherche, IRIF, Université Paris-Diderot	Président
Maurice Herlihy Professeur, Université de Brown	Rapporteur
Hans van Ditmarsch Professeur, *LORIA*	Rapporteur
Bernadette Charron-Bost Directrice de Recherche, LIX, École Polytechnique	Examinatrice
Yoram Moses Professeur, Technion – Israel Institute of Technology	Examinateur
Éric Goubault Professeur, LIX, École Polytechnique	Directeur de thèse
Samuel Mimram Professeur, LIX, École Polytechnique	Co-Directeur de thèse

Remerciements

Je remercie tout d'abord Sam et Éric, mes deux directeurs de thèse. Vous avez su me guider dans mes recherches, tout en me laissant la liberté d'explorer mes propres idées. Merci Éric pour ta disponibilité à toute épreuve, malgré tes responsabilités au sein du laboratoire. Je ne compte plus les e-mails reçus de ta part le dimanche à 23h. Merci Samuel pour tes conseils de rédaction, et pour nos nombreuses discussions scientifiques. Tu m'as appris à toujours chercher les adjonctions et à exhiber les propriétés universelles. Merci à vous deux pour toutes les connaissances que vous avez partagées avec moi.

Je remercie également mes deux rapporteurs, Hans et Maurice. Je suis infiniment reconnaissant pour votre relecture minutieuse de ce manuscrit. Merci pour la pertinence de vos remarques sur mon travail, et pour l'intérêt que vous portez à mes recherches. Merci aussi aux autres membres du jury, Bernadette, Pierre et Yoram, de m'accorder un peu de votre temps et de votre expertise.

Merci tout particulièrement à Sergio Rajsbaum pour notre collaboration fructueuse sur la logique épistémique. Nos multiples discussions tout au long de ma thèse ont grandement contribué à faire avancer mon travail. Merci de m'avoir permis de découvrir le Mexique, sa culture et sa cuisine. Merci aussi à Marijana qui m'a accompagné dans cette aventure mexicaine, et dans l'article qui en a résulté.

Je remercie chaleureusement toutes les autres personnes avec qui j'ai eu la chance de discuter de sujets scientifiques, lors des diverses rencontres et conférences auxquelles j'ai participées. Je pense notamment à Uli Fahrenberg, Martin Raussen, Krzysztof Ziemiański, Armando Castañeda, Petr Kuznetsov, et bien d'autres. Merci également aux professeurs qui m'ont donné le goût de l'informatique théorique, de la sémantique et de la théorie des catégories : Daniel Hirschkoff, Paul-André Melliès, et tout particulièrement Emmanuel Haucourt, sans qui je ne me serais sans doute pas tourné vers ce sujet de thèse. Merci aussi à mes propres élèves, avec qui j'ai découvert mon intérêt pour l'enseignement.

Merci à tous les membres du labo, avec qui j'ai partagé bureaux, pauses café et repas du midi : Adina, Aurélien, Benjamin, Bibek, Cameron, Emmanuel, François, Franck, Jérémy, Maxime, Nicolas, Patrick, Pierre-Yves, Robin, Sergio, Simon, Sylvie, Thibaut, Uli. Merci pour vos discussions amicales et votre bonne humeur. Je vous dois les souvenirs chaleureux de mon passage au LIX. Un grand merci également aux gestionnaires, Évelyne, Vanessa et Catherine, pour leur travail et leur efficacité.

Merci à tous mes amis de l'ENS Lyon et d'avant. Armaël et Pierre, avec qui j'ai partagé les galères de la rédaction de thèse ; Antoine, Cyprien, Tito, et les autres réguliers d'IRC ; et mes amis de prépa, Thomas, Paul, Palama, Térence, Bilou, Gaëtan, Lama, Nicolas, Ioum, Dimitri, Johan, Bastien. Mon passage à Fermat reste intégralement dans mon cœur.

Merci à ma famille, Maman, Papa, Violaine et Nico, qui ont su respecter mes moments de vacances en n'évoquant qu'avec parcimonie le sujet délicat de l'avancement de ma thèse. Et enfin, merci Marie, merci d'exister, tout simplement.

À Martin, avec qui j'ai publié mon tout premier article.
J'aurais été très fier de pouvoir te faire lire cette thèse,
C'est avec grand honneur que je te la dédie.

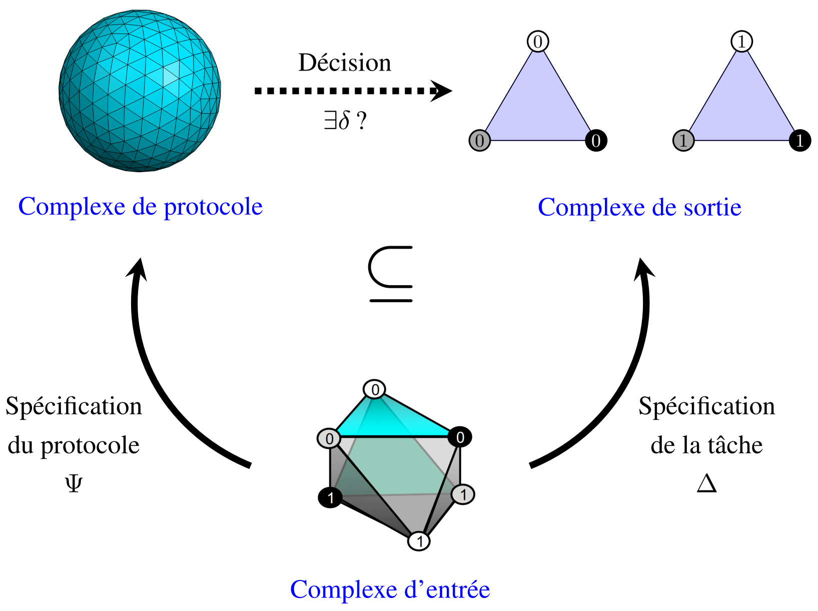
Contents

Résumé en Français

Tout système informatique distribué est sujet à des *pannes*. Le type de panne le plus courant est celui des systèmes par *passage de message*, où des ordinateurs distants doivent communiquer entre eux pour parvenir à accomplir une tâche en commun. Par exemple, lorsqu'une personne veut faire une transaction avec un site de vente en ligne, une communication se produit entre l'ordinateur personnel de l'utilisateur, les serveurs du site internet, et ceux de la banque. Les trois ordinateurs doivent soit accepter d'effectuer la transaction, soit la refuser (si l'utilisateur a un solde insuffisant sur son compte bancaire, par exemple). L'important est que les trois machines doivent réussir à se mettre d'accord : il ne faudrait pas que le compte bancaire de l'utilisateur soit débité alors que le site de vente n'a pas confirmé la transaction. Dans ce contexte, les canaux de communication que les ordinateurs utilisent pour transmettre des informations peuvent présenter un comportement peu fiable en raison des limitations physiques du matériel. En pratique, il peut arriver qu'un message envoyé d'un ordinateur à un autre soit retardé, perdu, ou même altéré, ce qui entraîne un comportement imprévisible. Même dans une architecture à mémoire partagée, où plusieurs processeurs se trouvent dans un seul système informatique, différentes sortes de pannes peuvent se produire. Par exemple, un processus peut subir une panne au cours de son exécution et s'arrêter subitement (une panne *crash*) ; ou même présenter un comportement arbitrairement malveillant (on parle alors de panne *Byzantine*). Comme ce type de comportement indésirable est inévitable, il doit être pris en compte lors de la conception de logiciels pour les systèmes distribués. Le domaine des *protocoles tolérants aux pannes* s'intéresse à la conception d'algorithmes fonctionnant sur diverses architectures de calcul distribué, en présence de défaillances. L'objectif d'un protocole tolérant aux pannes est d'aboutir à un résultat correct, même si certains composants du système tombent en panne. Le calcul distribué tolérant aux pannes est encore un domaine très actif, en particulier avec l'essor récent de la *blockchain* qui repose sur des variantes répétées du consensus, une tâche centrale en informatique distribuée.

Il est notoirement difficile de concevoir des algorithmes pour les systèmes distribués, d'autant plus en présence de pannes. En fait, dans certains modèles de calcul habituellement considérés, les tâches importantes deviennent parfois *impossibles* à résoudre. Le résultat séminal dans ce domaine a été établi en 1985 par Fischer, Lynch et Paterson [42], qui prouve que la *tâche du consensus* n'est pas résoluble dans un système par passage de messages avec au plus un crash potentiel. Plus tard, Biran, Moran et Zaks [10] ont proposé une caractérisation combinatoire de toutes les tâches concurrentes qui peuvent être résolues dans ce modèle. Leur critère repose sur un graphe « d'incertitude » des processus, dont la connexité est l'ingrédient principal de l'impossibilité. Il s'agit là du premier cas d'une caractérisation topologique (unidimensionnelle) de la résolubilité des tâches. Peu de temps après, il a été découvert que lorsque plus d'un processus peut planter, l'irrésolubilité des tâches est liée à des propriétés topologiques de plus haute dimension. Dans deux articles de 1993 [14, 112], cette découverte a été motivée par l'étude d'une tâche appelée *k-set agreement*, une forme de consensus plus faible où les processus doivent se mettre d'accord sur au plus k valeurs distinctes. Un troisième article indépendant de Herlihy et Shavit [70] a donné une caractérisation topologique de toutes les tâches qui sont résolubles dans un cadre asynchrone et *wait-free*, c'est-à-dire où l'on suppose qu'un nombre quelconque de processus peut tomber en panne. Cet article a introduit la notion de *complexe de protocole*, un complexe simplicial de dimension supérieure décrivant toutes les exécutions possibles qui pourraient se produire dans un protocole donné. En utilisant cette notion, ils ont formulé le *Théorème de Calculabilité Asynchrone*, qui dit que la résolution d'une tâche concurrente équivaut à l'existence d'une application simpliciale allant du complexe de protocole dans un *complexe de sortie* spécifiant la tâche qui nous intéresse. Ainsi, à travers ce théorème, la notion calculatoire de résolubilité d'une tâche est réduite à une question topologique, qui peut alors être abordée en utilisant les outils mathématiques usuels tels que la topologie algébrique ou la topologie combinatoire.

Cette caractérisation topologique de la résolubilité des tâches a été établie dans le contexte de processus asynchrones communiquant à travers des registres de lecture/écriture partagés (ou, de manière équivalente, des objets *immediate-snapshot* [15]). Mais la mémoire partagée ne constitue que le premier niveau dans la hiérarchie wait-free [63], et beaucoup d'autres objets avec des puissances de calcul variables méritent d'être considérés. Citons par exemple les objets *test-and-set* et *compare-and-swap*. Après quelques années, il est devenu clair que l'approche topologique de Herlihy et Shavit est également très efficace pour modéliser des processus communiquant par le biais de divers types d'objets partagés : les objets test-and-set dans [66], le passage de messages synchrone dans [68], les objets set-agreement ainsi que l'objet renommage dans [44]. Un compte rendu détaillé de ces résultats se trouve dans le récent livre de Herlihy, Kozlov et Rajsbaum [64]. Dans ce livre, la caractérisation topologique de la résolubilité des tâches (qui résulte du Théorème de la Calculabilité Asynchrone dans le cas des registres de lecture/écriture) devient maintenant une *définition* au lieu d'un théorème.



Fondamentalement, dans ce contexte, un « protocole » est spécifié par une *carrier-map* allant du complexe d'entrée vers le complexe de protocole. De même, une « tâche » est spécifiée par une carrier-map allant du complexe d'entrée vers le complexe de sortie. Enfin, on dit qu'un tel protocole *résout* une tâche, par définition, lorsqu'il existe une application simpliciale appelée « décision » allant du complexe de protocole au le complexe de sortie qui est incluse dans la spécification de la tâche. Le point fort de cette définition abstraite est qu'elle permet de restreindre autant que possible les raisonnements spécifiques au modèle. En effet, cette approche donne des résultats très généraux, mettant en relation les propriétés topologiques du complexe de protocoles (telles que k-connectivité, ou le fait d'être une pseudo-variété) avec la résolubilité de diverses tâches concurrentes. De telles techniques permettent de prouver des résultats d'impossibilité pour de vastes classes de protocoles en une seule fois, sans avoir à les considérer un à un.

Cette approche de modélisation du comportement des programmes par des objets mathématiques abstraits est évocatrice de la *sémantique dénotationnelle* [114] de Scott et Strachey. La sémantique dénotationnelle est généralement étudiée dans le contexte de programmes séquentiels, et tout particulièrement dans l'analyse des langages de programmation fonctionnels. Habituellement, on donne d'abord à un langage de programmation une *sémantique opérationnelle*, qui est une description concrète de la façon dont l'état interne du système évolue pendant l'exécution du programme. En revanche, la sémantique dénotationnelle est une description statique un programme, en lui associant un objet mathématique abstrait. L'exemple le plus simple de sémantique dénotationnelle est le modèle ensembliste, ou bien des variantes munis de davantage de structure ; mais d'autres modèles tels que la *sémantique de jeu* [2, 75] interprètent les programmes comme des stratégies dans un jeu à deux joueurs entre le programme et son environnement. Le choix d'une interprétation particulière d'un langage de programmation permet d'abstraire certaines propriétés non-essentielles des programmes, afin de mettre l'accent sur d'autres aspects qui nous intéressent. Enfin, une notion centrale en sémantique des langages de programmation est celle de la *compositionnalité* : un programme est généralement construit de manière modulaire, en assemblant des blocs de base pour créer un système plus complexe. Une bonne notion de sémantique des programmes doit être à même de refléter cette structure compositionnelle.

Dans cette thèse, nous adoptons un point de vue sémantique sur l'approche topologique pour les protocoles tolérants aux pannes. Notre premier objectif sera d'extraire ces constructions géométriques abstraites à partir d'une sémantique opérationnelle concrète pour la résolubilité des tâches. En effet, dans certains modèles de calcul, il peut être extrêmement difficile de définir correctement le complexe de protocole correspondant. Cela est dû au fait que le complexe de protocole décrit simultanément toutes les exécutions possibles du programme. Ce type de spécification du comportement d'un programme est rapidement sujet aux erreurs dès que l'on considère des systèmes un peu trop grands. De plus, ce point de vue n'est pas bien adapté pour analyser la composition de protocoles. Au lieu de décrire la totalité du protocole en un seul bloc, nous aimerions le décomposer en fonction de la sémantique de chacun des objets partagés qu'il utilise. Ainsi, la première étape est de définir une notion opérationnelle de *spécification concurrente*, définissant le comportement des objets partagés qu'un programme a le droit d'utiliser. Ensuite, nous devons construire un modèle de calcul à partir de ces spécifications concurrentes. En faisant cela, nous serons en mesure de donner un sens concret à ce que signifie pour un protocole de *résoudre* une tâche. La caractérisation topologique de la résolubilité d'une tâche peut alors être prouvée formellement à partir de cette sémantique opérationnelle. Cette direction de recherche nous mènera à prouver une version généralisée du Théorème de la Calculabilité Asynchrone, qui fonctionne pour n'importe quel objet partagé.

Notre deuxième objectif est de comparer cette sémantique topologique avec d'autres sémantiques pour les programmes concurrents. Une approche très importante pour l'étude des systèmes distribués est celle de Halpern et Moses [58, 106], qui est basée sur la *logique épistémique*, une logique modale permettant de raisonner à propos de la notion de *connaissance*. Des liens entre l'approche topologique et l'approche épistémique dans l'étude des protocoles tolérants aux pannes ont été soupçonnés depuis longtemps [112]. Néanmoins, aucun lien entre les deux domaines n'a été établi formellement. Dans cette thèse, nous montrons une correspondance étroite entre les complexes simpliciaux chromatiques qui sont utilisés pour étudier la calculabilité asynchrone, et les modèles de Kripke sur lesquels les formules de logique épistémique sont interprétées. Plus précisément, il y a une équivalence des catégories reliant les deux structures. Plus surprenant encore, les *carrier maps* qui modélisent les tâches et les protocoles dans [64] ont également un pendant dans le domaine de la *Logique Épistémique Dynamique (DEL)* [27]. DEL a été utilisée auparavant pour modéliser des systèmes distribués [12, 116], mais, à notre connaissance, elle n'a jamais été utilisée pour la calculabilité asynchrone. Cette correspondance a des conséquences importantes à la fois pour les logiciens et pour les informaticiens. Du point de vue de la logique épistémique, cette correspondance révèle la structure géométrique qui est se cache dans les modèles de Kripke. Inversement, du point de vue du calcul distribué, le lien avec la Logique Epistémique Dynamique donne un nouveau type de sémantique pour la calculabilité asynchrone, basée sur la logique. Cela permet de donner un sens plus concret aux preuves abstraites d'impossibilité, en termes des connaissances que les processus doivent acquérir pour résoudre une tâche.

Enfin, le troisième objectif de cette thèse sera de comparer la sémantique topologique de Herlihy et Shavit avec une autre sémantique géométrique pour les programmes concurrents : celle basée sur les *espace dirigés* de Fajstrup, Goubault, Raussen et. al [36]. Bien que ces deux sémantiques s'appuient sur les mêmes outils mathématiques, à savoir la topologie algébrique, elles le font de manière totalement différente. En effet, alors que le complexe de protocole encode les informations locales recueillies par les processus à la fin d'une exécution, la sémantique dirigée s'intéresse à l'étude des différents chemins par lesquels un processus peut évoluer d'un état initial à un état final. L'idée de base est la suivante : un programme est interprété comme un espace topologique doté d'une notion de direction, modélisant le passage du temps. Un *chemin* dans cet espace correspond à une exécution d'un programme, et deux chemins sont *homotopes* lorsqu'ils correspondent à des exécutions équivalentes. Les applications de ce point de vue comprennent, entre autres, la détection des interblocages [37] et la réduction de l'espace d'états [35]. La sémantique dirigée n'a encore jamais été utilisée pour étudier la calculabilité des tâches asynchrones. Dans cette thèse, nous ne donnons que des résultats préliminaires mettant en relation ces deux approches topologiques. A savoir, nous montrons comment diverses notions combinatoires de chemins dans les *Automates de Dimension Supérieure* [119, 34] sont liées à la sémantique opérationnelle basée sur les traces que nous avons développé dans le Chapitre 2 de cette thèse. Cette correspondance devrait être le point de départ de travaux futurs établissant des liens plus profonds entre la sémantique dirigée et celle basée sur les complexes de protocoles.

Plan de la thèse

Dans le chapitre 1, nous commençons par rappeler les définitions de la topologie combinatoire, ainsi que la façon dont elles sont utilisées dans [64] pour modéliser les tâches et les protocoles. Nous concluons le chapitre par une étude de cas, qui est une nouvelle contribution de cette thèse, en étudiant la résolubilité d'une tâche appelée *négation de l'égalité*.

Le chapitre 2 présente notre sémantique opérationnelle pour des processus asynchrones communiquant via des objets partagés. La contribution principale du chapitre est le Théorème de Calcul Asynchrone Généralisé (Théorème 2.73), qui montre comment la sémantique topologique peut être dérivée de notre sémantique opérationnelle concrète. En cours de route, nous discutons de diverses notions de spécifications d'objets concurrents basées sur la linéarisabilité et ses variantes. Enfin, nous présentons quelques travaux en cours concernant la composition des protocoles, et les liens avec la sémantique des jeux.

Le chapitre 3 est consacré au lien entre les complexes de protocoles et la logique épistémique. Nous commençons par rappeler les notions habituelles de logique épistémique basées sur les modèles de Kripke, puis nous prouvons dans le Théorème 3.20 que les complexes simpliciaux chromatiques peuvent être utilisés de façon équivalente comme modèles de logique épistémique. Nous explorons ensuite les conséquences de ce théorème, en traduisant les résultats habituels du calcul distribué dans le langage de la logique épistémique. Enfin, pour exhiber certaines limites à cette approche épistémique, nous étudions à nouveau la tâche de négation de l'égalité.

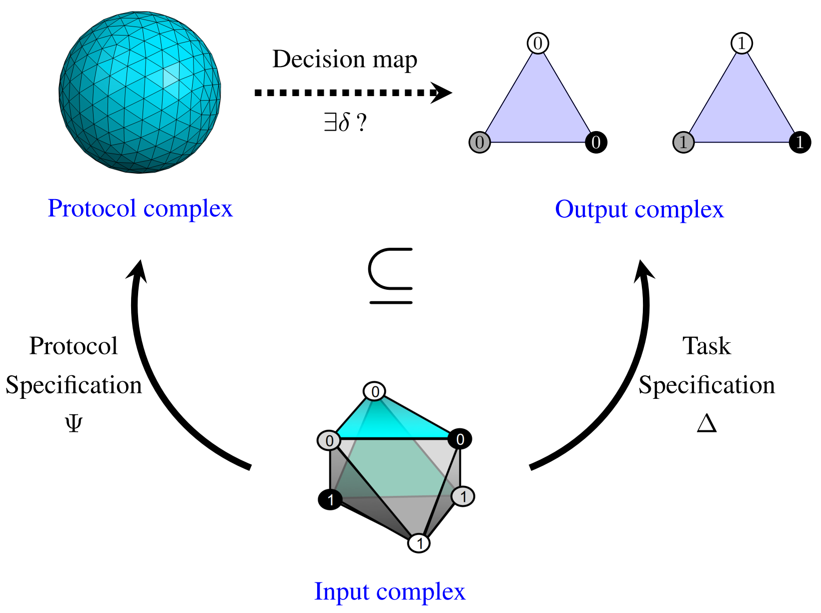
Enfin, dans le chapitre 4, nous étudions les liens avec sémantique dirigée. Nous définissons d'abord une version combinatoire des espaces dirigés, basée sur des ensembles pré-cubiques, et nous rappelons les différentes notions de chemins sur ces structures que l'on peut trouver dans la littérature. Ensuite, nous établissons un lien entre les chemins dans les espaces dirigés et les différentes notions de linéarisabilité que nous avons étudiées au le Chapitre 2.

Introduction

Every distributed computing system is subject to *failures*. The most obvious kind of failure can be found in the setting of message-passing systems, where distant computers need to communicate with each other in order to reach some kind of agreement. For instance, when a user wants to make an online transaction with an e-commerce website, a communication occurs between the user's personal computer, the website's servers and the user's bank. The three computers should either all agree to perform the transaction, or all agree to deny it (if the user has insufficient balance on their bank account). In this context, the communication channels that the computers use to transmit information can exhibit unreliable behavior due to the physical limitations of the hardware. Because of random perturbations in the physical transmission medium, a message which is sent from one computer to another might be delayed, lost, or even altered, resulting in unpredictable behavior. Even in a shared-memory architecture, where several processing units lie within a single computer systems, similar kinds of failures may occur. For example, a process may crash in the middle of a computation; it may crash before the computation starts, and not participate in it; or it may exhibit arbitrarily malicious behavior. Since this kind of unwanted behavior is inevitable, it must be taken into account when designing software for distributed systems. The field of *fault-tolerant distributed computing* is concerned with designing algorithms running on various distributed computing architectures, in the presence of failures. The goal of a fault-tolerant protocol is to be resilient to failures: even if some components of the system happen to crash, the remainder of the computing system should exhibit well-defined behavior. Fault-tolerant distributed computing is still a very active area, in particular with the recent rise of blockchain technology which is based on continuous variants of the consensus task.

It is notoriously difficult to design algorithms for distributed systems, even more so in the presence of failures. In fact, depending on the computational model that we consider, it might even become *impossible* to solve some concurrent tasks. The seminal result in this field was established in 1985 by Fischer, Lynch and Paterson [42], who proves that the *consensus task* is not solvable in a message-passing system with at most one potential crash. Later on, Biran, Moran and Zaks [10] came up with a combinatorial characterization of all the concurrent tasks which can be solved in that model. Their criterion relies on a graph of "process uncertainties", whose connectedness is the main ingredient for impossibility. This was the first instance of a (one-dimensional) topological characterization of task solvability. Soon thereafter, it was discovered that when more than one process may crash, the unsolvability of tasks is related to higher-dimensional topological properties. In two 1993 papers [14, 112], this discovery was motivated by the study of the k*-set agreement task*, a weaker form of consensus where the processes must agree on at most k different values. A third independent paper by Herlihy and Shavit [70] gave a topological characterization of all the tasks which are solvable in an asynchronous wait-free setting, that is, with any number of crashing processes. This paper introduced the notion of *protocol complex*, a higher-dimensional simplicial complex describing all the possible executions that might occur in a given protocol. Using this notion, they formulated the so-called *Asynchronous Computability Theorem*, which says that solving a concurrent task amounts to the existence of a simplicial map from the protocol complex to some *output complex* specifying the task that we are interested in. Thus, through this theorem, the computational notion of task solvability is reduced to a topological question, which can be tackled using classic mathematical tools such as algebraic topology or combinatorial topology.

This topological characterization of task solvability was established in the context of asynchronous processes communicating through shared read/write registers (or, equivalently, immediate-snapshot objects [15]). But read/write registers are only the first level in the wait-free hierarchy [63], and many other objects with varying computational power are worth considering, such as test-and-set or compare-and-swap. Over the years, it became clear that the protocol complex approach of Herlihy and Shavit is also very successful to model processes communicating through various kinds of shared objects: test-and-set objects in [66], synchronous message-passing in [68], set-agreement objects and renaming objects in [44]. A thorough account of such results can be found in the recent book by Herlihy, Kozlov and Rajsbaum [64]. In this book, the topological characterization of task solvability (given by the Asynchronous Computability Theorem in the case of read/write registers) now becomes a *definition* instead of a theorem. Basically, in this context, a "protocol" is formally specified as a *carrier-map* from the input complex to some protocol complex. Similarly, a "task" is specified as a carrier map from the input complex to the output complex. Finally, we say that a protocol *solves* a task, by definition, when there exists a particular simplicial map from the protocol complex to the output complex. Their motivation for using such an abstract definition of protocols and solvability is that they want to "*restrict model-specific reasoning as much as possible*". Indeed, this approach yields very general results, relating topological properties of the protocol complex (such as k-connectedness, or being a pseudomanifold) to the solvability of various concurrent tasks. Such techniques allow us to prove impossibility results for wide classes of protocols at once, instead of considering them one at a time, provided that their protocol complexes have similar topological properties.



This approach of modeling program behavior by abstract mathematical objects is evocative of Scott and Strachey's *denotational semantics* [114]. However, to our knowledge, it has never been studied from that perspective. Denotational semantics are usually studied in the context of sequential programs, and have been particularly successful at analyzing functional programming languages. Usually, a programming language is first given an *operational semantics*, which is a concrete description of how the internal state of the system evolves while the program is being executed. In contrast, denotational semantics give a static, high-level description of a program by associating it with an abstract mathematical object. The simplest kind of denotational semantics interprets programs as set-theoretic functions, usually with additional structure; but others such as *game semantics* [2, 75] interpret programs as strategies in a two-player game between a program and its environment. By interpreting programs as mathematical objects, we lose some information about them: some computational properties of the programs are abstracted away, in order to emphasize other aspects of the programs. The goal of denotational semantics is to build adequate mathematical abstractions that allow us to focus on one particular aspect of the programming language that we want to study. Finally, a central notion in program semantics is that of *compositionality*: computer programs are usually built in a modular way, by assembling basic components to create a complex system. A good notion of program semantics should be able to reflect this compositional structure.

In this thesis, we propose to take a semantical perspective on the protocol complex approach to asynchronous computability. Our first goal will be to ground it in a concrete operational semantics for task solvability. Indeed, for some intricate computational models, it can be extremely difficult to produce their corresponding protocol complex. Essentially, one has to describe simultaneously in one huge combinatorial object, all the possible executions of the protocol, and how they are linked together with respect to the local views of the processes. This kind of specification of a protocol complex is very error-prone for large systems, and it is not well adapted for composing protocols. Instead, we would like to derive the protocol complex from the formal semantics of each of the shared objects that it is using. So, the first step is to define an operational notion of *concurrent object specification*, defining the behavior of the shared objects that a protocol is allowed to use. Then, we need to define what it means to compute using these objects, i.e., build a computational model out of these concurrent specifications. By doing so, we will be able to give a concrete meaning to what it means for a protocol to *solve* a task. The topological characterization of task solvability can then be derived from this operational semantics: what we obtain is a generalized version of the Asynchronous Computability Theorem, which works for arbitrary objects instead of read/write registers.

Our second goal is to compare the protocol complex semantics with other semantics for concurrent programs. One very important approach for studying distributed systems is that of Halpern and Moses [58, 106], which is based on *epistemic logic*, the modal logic of knowledge. It has been remarked from the very beginning of the topological approach to task solvability that the protocol complex approach has an intuitive explanation in terms of the *knowledge* that the agents need to gain in order to solve a task [112]. But, to the best of our knowledge, no formal link has been established between the two fields. We exhibit a close correspondence between the chromatic simplicial complexes that are used to study asynchronous computability, and the Kripke models on which epistemic logic formulas are interpreted; namely, there is an equivalence of categories relating the two structures. More surprisingly, the carrier maps which model tasks and protocols in [64] also have a counterpart in the field of *Dynamic Epistemic Logic (DEL)* [27], the so-called *product-update model* construction. This correspondence has important consequences in both fields. From the point of view of epistemic logic, this connection uncovers the higher-dimensional information which is already present in Kripke models, but which is hidden by the usual graph-based formalism. The connection between the consensus task, common knowledge and connectedness is well known to epistemic logicians [29]; but no notion of knowledge has ever been shown to correspond to higher-dimensional connectedness properties. Well-understood examples from distributed computing, such as k-set-agreement, are a good starting point to understand the meaning of such topological invariants in terms of knowledge. Conversely, from a distributed computing perspective, the link with Dynamic Epistemic Logic gives a new kind of semantics for asynchronous computability, based on logic: one can now specify a task directly as an epistemic logic formula, expressing the knowledge that the agents should manage to acquire. Even though DEL has been used before to model distributed systems [12, 116], as far as we know it has never been used in the context of asynchronous computability.

Finally, the third goal of this thesis will be to compare the protocol complex approach with another geometric semantics for concurrent programs: the so-called *directed space semantics* of Fajstrup, Goubault, Raussen et. al [36]. Although these two semantics rely on the same mathematical tools, namely, algebraic topology, they do so in quite different ways. Indeed, while the protocol complex encodes the local information gathered by the processes at the end of an execution, the directed space semantics is concerned with studying the different paths by which a process can evolve from an initial state to a final state. The basic idea is the following: a program is interpreted as a topological space equipped with a notion of direction, modeling the passage of time. A *path* in this space corresponds to an execution of a program, and two paths are *homotopic* when they correspond to equivalent executions. Applications of this point of view include, among others, deadlock detection [37] and state-space reduction [35]. It has not been used before to study asynchronous task computability. In this thesis, we only give preliminary results relating this topological approach with the protocol complex semantics. Namely, we show how various combinatorial notions of paths in *Higher-Dimensional Automata* [119, 34] are related to the trace-based operational semantics that we develop in the first part of this thesis. This correspondence should be the starting point of future work establishing deeper links between the directed space and protocol complex semantics.

Plan of the thesis

In Chapter 1, we start by recalling the definitions of combinatorial topology, and how they are used in [64] to model protocols, tasks, and task solvability. We conclude the chapter with a case study, which is a novel contribution of this thesis, by studying the wait-free solvability of the so-called *equality negation task*.

Chapter 2 introduces our operational semantics for asynchronous processes communicating through shared objects. The main contribution of the chapter is our Generalized Asynchronous Computability Theorem (Theorem 2.73), which shows that the protocol complex semantics can be derived from our concrete operational semantics. Along the way, we discuss various notions of specifications for concurrent objects based on linearizability, and compare them with our definition. Finally, we present some work in progress concerning the compositionality of protocols, and links with game semantics.

Chapter 3 is dedicated to the link between protocol complexes and epistemic logic. We start by recalling the usual notions of epistemic logic based on Kripke models, then we prove in Theorem 3.20 that chromatic simplicial complexes can equivalently be used as models for epistemic logic. We then explore consequences of this theorem, by translating usual results of distributed computing in the language of epistemic logic. We find some limitations of this epistemic logic approach by studying the equality negation task, and we discuss possible ways to fix these limitations and their relevance.

Finally, in Chapter 4, we study the relationship between protocol complex semantics and directed space semantics. We first define a combinatorial version of directed spaces, based on pre-cubical sets, and we recall the various notions of paths on these structures that can be found in the literature. Then, we show how they are related with the variants of linearizability that we studied in Chapter 2, and we prove that the *standard chromatic subdivision* can be recovered as the space of paths in a cube.

Acknowledgement

Some of the pictures which are used in this thesis to illustrate 3-dimensional protocol complexes were taken from articles of Maurice Herlihy and Sergio Rajsbaum [65, 69], with permission from the second author, as well as from the book by Herlihy, Kozlov and Rajsbaum [64]. One picture of a triangulated sphere is a modified version of the one used in the Wikipedia article “Geodesic Polyhedron”[1], under the Creative Commons Attribution-Share Alike 4.0 International license.

[1]https://commons.wikimedia.org/wiki/File:Geodesic_icosahedral_polyhedron_example.png

Chapter 1

Preliminaries

In this chapter, we recall the important notions behind the simplicial complex model of distributed computability. Most of the definitions and theorems presented here can be found in the book by Herlihy, Kozlov and Rajsbaum [64], except for Section 1.4 which is novel work, recently accepted for publication [50].

In Section 1.1, we discuss informally the distributed computing setting, and the kind of problems that we want to study. Section 1.2 introduces the notions from combinatorial topology that we will need in the remainder of the thesis, as well as two important lemmas, Sperner's lemma and the Index lemma. In Section 1.3, we show how the distributed computing concepts of tasks and protocols can be modeled using combinatorial topology. Finally, in Section 1.4, we showcase how this topological characterization of task solvability can be used to prove impossibility results. In particular, we propose to study the so-called *equality negation* task, whose unsolvability hinges on subtle topological properties.

1.1 The distributed computing setting

The field of *fault-tolerant computability* studies which *concurrent tasks* can or cannot be solved in various computational models. In particular, the topological approach that we present here has been developed in order to prove *impossibility results*. One of the first such result is known as "FLP impossibility" [42], and says that in an asynchronous message-passing system with one faulty process, the consensus task is not solvable. Soon thereafter, people started developing powerful mathematical tools based on algebraic topology in order to prove impossibility results [14, 112, 70].

There is a wide variety of distributed computing models that we might want to consider:

– the processes might be synchronous or asynchronous;
– we might consider shared memory or message-passing systems;
– various shared objects or synchronization primitives might be available to the processes;
– there are various kinds of faults, from simple crashes where a faulty process just stops computing, to messages getting lost, or byzantine failures where a process can send false information.

Unlike in sequential computability theory, where Turing machines provide a universal computing model, all of these models are worth studying, and they can vary greatly in terms of computational power.

1.1.1 Tasks

In the basic setting that we study here, a fixed number n of processes are running concurrently. Initially, each process is given a private input value. After communicating with each other using the communication primitives at their disposal, each process produces an output value. The goal of such a computation is to satisfy some relationship between the inputs and the outputs; this relation is called the *task specification*. Tasks have been studied since early on in distributed computability [10].

The most well-known task is the *consensus task*, where the participating processes must agree on one of their input values. For instance, if three processes start with input values $(1, 2, 3)$, their goal is to produce the same output, either $(1, 1, 1)$, or $(2, 2, 2)$, or $(3, 3, 3)$. More precisely, if the n processes start the computation with input values $(v_1, \dots, v_n)$, they should output a set of decision values $(d_1, \dots, d_n)$ such that the following two properties are satisfied:

– *Validity*: each decision is among the proposed inputs, i.e., for all i, there exists j such that $d_i = v_j$.
– *Agreement*: all processes decide on the same output, i.e., $d_1 = \cdots = d_n$.

In fact, to account for the possibility of crashing processes, one should also specify what the expected behavior is when some processes are missing. The fully formal definition of a task will be given in Section 1.3.1.

1.1.2 Processes, programs and protocols

In principle, the simplicial complex approach to distributed computing should be able to model most computational models that we can conceive of. However, in this thesis, we will usually restrict ourselves to the context of asynchronous processes communicating through shared objects. In this section, we discuss the various assumptions about the computational model that we usually make.

A fixed number of processes. A very common assumption in the field of fault-tolerant computability is that we are running a fixed number n of processes. Those n processes start the computation together, and once a process has decided its output value, it does not intervene in the computation anymore. The processes are not allowed to create new threads: so, each process has its own sequential *program*, and these n programs are running concurrently. A *protocol* simply consists of one program for each process.

The processes have unique identifiers. Each process has access to its own identifier, or “PID”, and all processes have distinct identifiers. Moreover, all the processes know in advance the names of all the other processes; in particular, they know the number n of participating processes. In practice, this assumption is enforced by the fact that we allow each process to have its own program. This is contrasted with the context of *anonymous* computation, where all the processes run the same program. In this context, the suitable notion of task is called a *colorless task* [64].

Processes are asynchronous. We will mostly be interested in asynchronous computation, where an execution consists of an arbitrary interleaving of operations of all the processes. However, in some cases, a round-based structure can be recovered in an asynchronous model, such as in the case of the *iterated immediate-snapshot* protocol [15].

Protocols are wait-free. We usually require the protocols to be *wait-free*, meaning that every process must be guaranteed to terminate its program in a bounded number of steps. In the presence of asynchrony, this implies that a process is not allowed to wait until it receives information from any of the other processes, since they may be arbitrarily slow. This assumption makes sense when we consider that any number of processes are allowed to crash during the computation. In contrast, the weaker t-*resilience* assumption is also often considered, when we consider that at most t processes can crash. Intuitively, that means that a process is allowed to wait until it hears from $n - t - 1$ other processes before making any progress. We will occasionally mention t-resilience in the rest of the thesis, but the focus will be mostly on wait-free protocols.

Faulty processes do not participate in the execution. The only kind of failure that we consider is when some processes crash before they take any steps in the computation. Thus, in an execution of a protocol, we simply select a subset of the n processes (the non-faulty ones), and we run their programs together. The subset of participating processes is not known in advance by the processes. This assumption is not as restrictive as it seems: indeed, in an asynchronous and wait-free context, a process that crashes in the middle of its execution is indistinguishable from a process that is just very slow. Moreover, other kinds of failures such as messages getting lost, or byzantine failures, can be encoded in the behavior of the communication primitives that we use (see Section 1.1.4).

1.1.3 The (un)importance of crashes

In many cases, the two ingredients that allow us to prove impossibility results are *asynchrony* and *wait-freeness*. Once we make those two assumptions, the possibility of crashing processes often becomes irrelevant: this is for example the case for the usual impossibility of solving consensus [63] and set-agreement [112] using read/write registers. Similarly, the impossibility result that we present in Section 1.4 takes place in a setting without any crashes: all the n processes are guaranteed to participate in the execution, and to run until the end of their program.

Of course, the reason why one would study wait-free protocols in the first place is that we want our programs to be correct in the presence of failures. But the point that we want to make here is that there is often no need to explicitly model the crashes in order to obtain impossibility results.

However, there is one subtle difference between whether or not we allow the processes to crash. We illustrate it by comparing the *consensus* task (see Section 1.1.1), and the *leader election* task. The leader election task is very similar to consensus, except that instead of being given input values, the processes must elect a leader, that is, agree on one of their PIDs. Thus, in a leader election task, every process knows in advance its "input", as well as the inputs of the other processes.

If there is no possibility of crashes, there is a trivial solution to the leader election task: every process elects the process with the smallest PID, without needing to communicate with the others. This protocol would not work if crashes can happen: indeed, if the process with the smallest PID has crashed (does not participate in the computation), then this is not a valid decision anymore. And in fact, leader election is unsolvable in an asynchronous wait-free setting using read/write registers, if there is a possibility of crash.

On the other hand, the consensus task, where the processes do not know their inputs in advance, is unsolvable whether or not there is a possibility of crashes. Indeed, consider a model without crashes, and suppose that a process P starts the computation with input i. Since our model is asynchronous, it might be the case that all the other processes are very slow; and since the model is wait-free, process P must eventually decide on an output after running in isolation. Since there is a possibility that the set of inputs was $(i, i, \dots, i)$, and because of the validity requirement, the only possible decision is to return i. This is precisely the restriction that we would add in a model with crashes: in a solo execution, a process must return its own input. Here, there is no need to add this requirement in advance, since it can be derived from the wait-free and asynchrony assumptions, and the task specification.

This distinction will be important in Chapter 3, where the epistemic logic definition of task solvability is naturally "without crashes", but we are still able to prove impossibility results.

1.1.4 Shared objects

As we mentioned before, in our setting, the processes communicate through *shared objects*. Here, the notion of object should be understood in an abstract sense; in particular, it does not necessarily mean shared memory. An *object* is seen as a "black box" with which the processes can interact by calling its *methods*. Examples of objects include:

- shared memory primitives: read/write registers, immediate-snapshot objects,
- synchronization primitives: test-and-set, compare-and-swap,
- concurrent data structures: lists, queues, trees, hashmaps,
- message-passing interfaces: reliable or with failures,
- consensus objects, set-agreement objects, ...

Chapter 2 will discuss thoroughly the various methods that can be used to specify the behavior of such objects. For the moment, the main point that we want to make is that all these objects can be specified by giving their interface (i.e., a set of methods), and by specifying the behavior of these methods. For instance, lists and queues both have the same interface (with methods `push` and `pop`), but depending on the case, the specification of the `pop` method will say that it returns either the last or the first value that was previously pushed. Similarly, a message-passing interface is usually specified by its methods `send` and `receive`, but depending on their specification, we can model reliable or unreliable operations.

Since we are interested in an asynchronous wait-free setting, our main degree of freedom to produce models of varying computational power will be the choice of objects that the processes have access to.

1.2 Combinatorial topology

In this section, we recall the definitions from combinatorial topology that we will be using in the rest of the thesis. A more thorough account of these notions can be found in [64].

1.2.1 Simplicial complexes and simplicial maps

The main basic structure that we will use is called a *simplicial complex*. Combinatorially, it can be seen as a kind of higher-dimensional generalization of a graph: it has vertices and edges, but also higher-dimensional cells such as triangles, tetrahedra, and so on. A n-dimensional cell is called a n-*simplex* or just *simplex* (plural *simplices*), when the dimension is irrelevant, or clear from the context. Thus, vertices, edges, triangles and tetrahedra can also be called 0, 1, 2 and 3-simplices, respectively. Geometrically, an n-simplex corresponds to the *convex hull* of $n + 1$ affinely independent points in $\mathbb{R}^d$, for $d \geq n$.

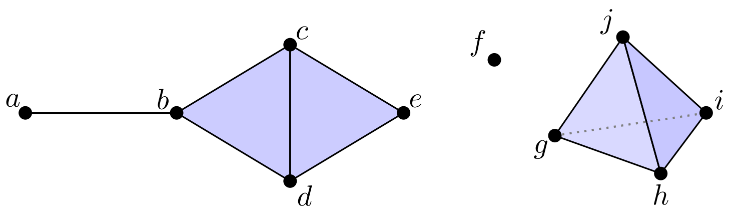
Although this geometric interpretation will be useful when we use pictures to illustrate what is going on in low-dimensional examples, our formalization will rely entirely on the following combinatorial definition. In the literature, the name *abstract simplicial complex* is often used to differentiate it from other more geometric definitions; here we simply refer to them as *simplicial complexes*, since it is the only definition that we use.

Definition 1.1. A *simplicial complex* $\mathcal{C} = (V, S)$ consists of a set V of *vertices* and a set S of non-empty finite subsets of V called *simplices*, such that:

– for each vertex $v \in V$, $\{v\} \in S$, and
– S is closed under containment, i.e., if $X \in S$ and $Y \subseteq X$ with $Y \neq \varnothing$, then $Y \in S$.

We sometimes abuse notations and write $X \in \mathcal{C}$ instead of $X \in S$ to mean that X is a simplex of $\mathcal{C}$. If $Y \subseteq X$, we say that Y is a *face* of X. The simplices that are maximal w.r.t. inclusion are called *facets*. The *dimension* of a simplex $X \in S$ is $\dim(X) = |X| - 1$, and a simplex of dimension n is called an n-simplex. We usually identify a vertex $v \in V$ with the 0-simplex $\{v\} \in S$. The dimension of the simplicial complex $\mathcal{C}$ is $\dim(\mathcal{C}) = \sup\{\dim(X) \mid X \in S\}$. A simplicial complex is *pure* if all its facets have the same dimension (which is also the dimension of the complex). We say that a simplicial complex $\mathcal{C} = (V, S)$ is a *subcomplex* of $\mathcal{C}' = (V', S')$ when $S \subseteq S'$, and we write it $\mathcal{C} \subseteq \mathcal{C}'$.

Example 1.2. The simplicial complex represented below has set of vertices $V = \{a, b, c, d, e, f, g, h, i, j\}$, and its five facets are $\{a, b\}$, $\{b, c, d\}$, $\{c, d, e\}$, $\{f\}$, and $\{g, h, i, j\}$. So, the set S of simplices contains all subsets of these facets. For instance, the 2-dimensional simplex $\{g, h, j\}$ is a face of $\{g, h, i, j\}$. This simplicial complex is of dimension 3 because the largest simplex $\{g, h, i, j\}$ is a 3-simplex.



Definition 1.3. A *simplicial map* $f : \mathcal{C} \to \mathcal{C}'$ from $\mathcal{C} = (V, S)$ to $\mathcal{C}' = (V', S')$ is a mapping $f : V \to V'$ between the vertices of the two complexes, such that the image of a simplex is a simplex, i.e., for every $X \in S$, we have $f(X) := \{f(v) \mid v \in X\} \in S'$.

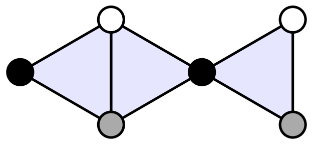
A simplicial map is *rigid* if the image of each simplex $X \in S$ has the same dimension, i.e., if $\dim(f(X)) = \dim(X)$. It is easily checked that the composition of two simplicial maps is a simplicial map, and that rigidity is preserved by composition. If $\mathcal{K} \subseteq \mathcal{C}$ is a subcomplex of $\mathcal{C}$, and $f : \mathcal{C} \to \mathcal{C}'$ is a simplicial map, we write $f(\mathcal{K}) = \bigcup_{X \in \mathcal{K}} f(X)$, which is a subcomplex of $\mathcal{C}'$.

Chromatic simplicial complexes. To model distributed computation, we will often decorate the vertices of our simplicial complexes with various labelings: process names, input and output values, local states. In some cases, it is important that in each simplex, all the vertices are assigned distinct labels. When this is the case, we usually refer to these labels as *colors*. When a simplicial complex is equipped with such a coloring, we say that it is a *chromatic* simplicial complex.

In the following, we fix a finite set A whose elements are called *colors*.

Definition 1.4. A *chromatic simplicial complex* $\mathcal{C} = (V, S, \chi)$ consists of a simplicial complex (V, S) equipped with a *coloring map* $\chi : V \to A$ such that all vertices of every simplex $X \in S$ have different colors, that is, for all $X \in S$, for all $v, v' \in X$, if $v \neq v'$ then $\chi(v) \neq \chi(v')$.

On the pictures, we often use the three colors black, gray and white, for printer-friendliness. The example below depicts a pure 2-dimensional chromatic simplicial complex with three colors.



Definition 1.5. A *chromatic simplicial map* $f : \mathcal{C} \to \mathcal{C}'$ from $\mathcal{C} = (V, S, \chi)$ to $\mathcal{C}' = (V', S', \chi')$ is a mapping $f : V \to V'$ between the vertices of the two complexes, such that:

– f is a simplicial map on the underlying simplicial complexes, and

– f is *color-preserving*, i.e., for every $v \in V$, $\chi'(f(v)) = \chi(v)$.

Note that a chromatic simplicial map is necessarily rigid, and moreover, the composition of two such maps is still a chromatic simplicial map.

Remark 1.6. An alternative way to define chromatic simplicial complexes is to consider the simplicial complex $\Delta^A = (A, \mathscr{P}(A))$, where $\mathscr{P}(A)$ is the powerset of A. Then, a chromatic simplicial complex is given by a simplicial complex $\mathcal{C} = (V, S)$ and a rigid simplicial map $\chi : \mathcal{C} \to \Delta^A$. A chromatic simplicial map $f : \mathcal{C} \to \mathcal{C}'$ is a simplicial map from $\mathcal{C}$ to $\mathcal{C}'$ that makes the following diagram commute:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\chi"] \arrow[d, "f"'] & \Delta^A \\
\mathcal{C}' \arrow[ur, "\chi'"'] &
\end{tikzcd}$$

In other words, the category of chromatic simplicial complexes and chromatic simplicial maps is the slice category $\mathsf{RigidSC}/\Delta^A$, where $\mathsf{RigidSC}$ is the category of simplicial complexes and rigid simplicial maps.

1.2.2 Carrier maps

To model distributed computing, we also need another kind of map between simplicial complexes, called a *carrier map*. A carrier map Φ associates with each simplex X of $\mathcal{C}$, a subcomplex $\Phi(X)$ of $\mathcal{C}'$; therefore, we write such a map $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$, in order to mimic the usual powerset notation.

Definition 1.7. Let $\mathcal{C} = (V, S)$ and $\mathcal{C}' = (V', S')$ be two simplicial complexes. A *carrier map* Φ from $\mathcal{C}$ to $\mathcal{C}'$, written $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$, assigns to each simplex $X \in S$ a subcomplex $\Phi(X)$ of $\mathcal{C}'$, such that Φ is monotonic, i.e., if $Y \subseteq X$ then $\Phi(Y) \subseteq \Phi(X)$.

Given two carrier maps $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$ and $\Psi : \mathcal{C} \to 2^{\mathcal{C}'}$, we say that Φ is *carried by* Ψ, written $\Phi \subseteq \Psi$, when for every simplex $X \in \mathcal{C}$, $\Phi(X) \subseteq \Psi(X)$. A carrier map is called *rigid* if for every simplex $X \in S$ with $\dim(X) = d$, the subcomplex $\Phi(X)$ is pure of dimension d. For a subcomplex $\mathcal{K} \subseteq \mathcal{C}$ of $\mathcal{C}$, we write $\Phi(\mathcal{K}) = \bigcup_{X \in \mathcal{K}} \Phi(X)$. Then the composition of two carrier maps can be defined as follows: given $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$ and $\Psi : \mathcal{C}' \to 2^{\mathcal{C}''}$, their composition is $\Psi \circ \Phi : \mathcal{C} \to 2^{\mathcal{C}''}$, defined by $\Psi \circ \Phi(X) = \Psi(\Phi(X))$. It is easily checked that $\Psi \circ \Phi$ is still a carrier map, and that if both Φ and Ψ are rigid, then so it $\Psi \circ \Phi$. It is also possible to compose a carrier map with a simplicial map. Let $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$ be a carrier map, and $f : \mathcal{C}' \to \mathcal{C}''$ a simplicial map. Then $f \circ \Phi : \mathcal{C} \to 2^{\mathcal{C}''}$, defined by $f \circ \Phi(X) = f(\Phi(X))$, is a carrier map. We could also define similarly the precomposition of a carrier map with a simplicial map, but actually we will only use the postcomposition.

Chromatic carrier maps. Unlike in the case of chromatic simplicial maps, here the rigidity is not implied by the preservation of colors. So, we also require that a chromatic carrier map should be rigid.

Definition 1.8. A *chromatic carrier map* Φ from $\mathcal{C} = (V, S, \chi)$ to $\mathcal{C}' = (V', S', \chi')$, written $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$, assigns to each simplex $X \in S$ a subcomplex $\Phi(X)$ of $\mathcal{C}'$, such that:

– Φ is monotonic, i.e., if $Y \subseteq X$ then $\Phi(Y) \subseteq \Phi(X)$,
– Φ is rigid, i.e., for every simplex $X \in S$ of dimension d, $\Phi(X)$ is pure of dimension d,
– Φ is color-preserving, i.e., for every $X \in S$, $\chi(X) = \chi'(\Phi(X))$, where $\chi(X) = \{\chi(v) \mid v \in X\}$ and $\chi'(\Phi(X)) = \bigcup_{Z \in \Phi(X)} \chi'(Z)$.

Given two chromatic carrier maps $\Phi : \mathcal{C} \to 2^{\mathcal{C}'}$ and $\Psi : \mathcal{C}' \to 2^{\mathcal{C}''}$, and a chromatic simplicial map $f : \mathcal{C}' \to \mathcal{C}''$, we can check that the compositions $\Psi \circ \Phi$ and $f \circ \Phi$ are still chromatic carrier maps.

1.2.3 The standard chromatic subdivision

An important operation on chromatic simplicial complexes is called the *standard chromatic subdivision*. This construction will naturally arise when we model the immediate-snapshot protocol complex in Section 1.3.2. Chromatic subdivisions have been thoroughly studied by distributed computer scientists, due to the fact that they are able to model wait-free computation using read/write registers [64, 54, 80].

We start with an abstract combinatorial description of this construction, without referring to the colors:

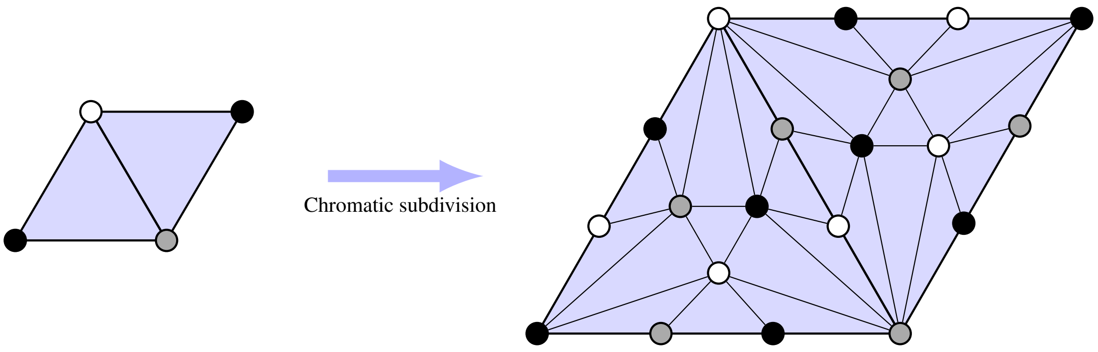
Definition 1.9. Let $\mathcal{C} = (V, S)$ be a simplicial complex. Its *standard chromatic subdivision* $\mathsf{ChSub}(\mathcal{C})$ has vertices of the form (v, X_v) where $v \in V$ is a vertex of $\mathcal{C}$, and $X_v \in S$ is a simplex of $\mathcal{C}$ that contains v. The set of vertices $\{(v_0, X_{v_0}), \dots, (v_k, X_{v_k})\}$ is a k-simplex if:

– it can be indexed so that $X_{v_0} \subseteq \ldots \subseteq X_{v_k}$, and
– for all i, j, if $v_i \in X_{v_j}$, then $X_{v_i} \subseteq X_{v_j}$.

Of course, as the name indicates, the most notable property of this subdivision operator is that when we subdivide a chromatic simplicial complex, we still have a chromatic simplicial complex.

Proposition 1.10. *Let $\mathcal{C} = (V, S, \chi)$ be a chromatic simplicial complex. Then $\mathsf{ChSub}(\mathcal{C})$, equipped with the coloring $\widehat{\chi}(v, X_v) := \chi(v)$, is a chromatic simplicial complex.*

Proof. Let $\{(v_0, X_{v_0}), \dots, (v_k, X_{v_k})\}$ be a simplex of $\mathsf{ChSub}(\mathcal{C})$. First, notice that the vertices $v_0, \dots, v_k$ are distinct: if $v_i = v_j$ for $i \neq j$, then since $v_i \in X_{v_j}$ and $v_j \in X_{v_i}$, we would have $X_{v_i} = X_{v_j}$. Moreover, if we index the vertices so that $X_{v_0} \subseteq \ldots \subseteq X_{v_k}$, then X_{v_k} is a simplex of $\mathcal{C}$ that contains all the vertices $v_0, \dots, v_k$. Since X_{v_k} is properly colored, all the vertices v_i must have distinct colors. $\square$



There is an associated carrier map $\Psi : \mathcal{C} \to 2^{\mathsf{ChSub}(\mathcal{C})}$, carrying each simplex of $\mathcal{C}$ to its subdivided image in $\mathsf{ChSub}(\mathcal{C})$. Formally, a simplex $X \in \mathcal{C}$ is sent to the subcomplex of $\mathsf{ChSub}(\mathcal{C})$ consisting of all the simplices $\{(v_0, Y_{v_0}), \dots, (v_k, Y_{v_k})\}$ such that for all i, $Y_{v_i} \subseteq X$. It is easily checked that when $\mathcal{C}$ is a chromatic simplicial complex, Ψ is a chromatic carrier map [64].

Finally, the *iterated chromatic subdivision* of $\mathcal{C}$ is obtained by iterating the previous construction, subdividing each simplex of $\mathsf{ChSub}(\mathcal{C})$ and so on. We denote by $\mathsf{ChSub}^k(\mathcal{C})$ the complex obtained by iterating the subdivision operation k times.

1.2.4 Pseudomanifolds with boundary

We now define a class of simplicial complexes with nice topological properties, called *pseudomanifolds with boundary* or simply *pseudomanifolds*. Our main motivation for introducing them is to be able to formulate two important theorems of combinatorial topology, specifically Sperner's lemma (Section 1.2.5 and the Index lemma (Section 1.2.6), both of which involve pseudomanifolds.

Definition 1.11. A pure simplicial complex $\mathcal{C} = (V, S)$ of dimension n is *strongly connected* if any two n-simplices can be connected by a sequence of n-simplices where two consecutive simplices share an $(n-1)$-dimensional face. More formally, for all $X, Y \in S$ of dimension n, there exists a sequence of n-simplices $X_0, \dots, X_k \in S$, such that $X_0 = X$, $X_k = Y$, and for all i, $\dim(X_i \cap X_{i+1}) = n - 1$.

Definition 1.12. A simplicial complex $\mathcal{C}$ is a *pseudomanifold with boundary* if:

- it is pure of dimension n,
- it is strongly connected,
- and every $(n-1)$-simplex is a face of either one or two n-simplices.

The set of $(n-1)$-simplices that are the face of exactly one n-simplex induces a pure $(n-1)$-dimensional subcomplex of $\mathcal{C}$, called the *boundary* of $\mathcal{C}$, and written $\partial\mathcal{C}$.

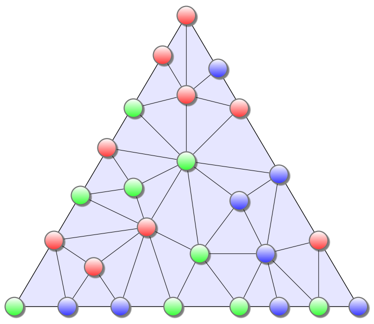
As the name indicates, pseudomanifolds with boundary are the combinatorial counterpart of a *manifold* in standard topology. However, they are less restricted than the usual manifolds, for two reasons. Firstly, they are allowed to have a boundary, corresponding to the $(n-1)$-simplices that are the face of exactly one n-simplex; secondly, they are allowed to have singularities or "pinches".

1.2.5 Sperner's lemma

Sperner's lemma is perhaps the most well-known result of combinatorial topology. It is a discrete version of Brouwer's fixed-point theorem, which says that a continuous function from the n-dimensional unit ball into itself must have a fixed point. The simplest formulation of Sperner's lemma deals with colorings of a subdivided triangle (or, in dimension n, a subdivided n-simplex); but in fact, it can be generalized to any pseudomanifold of dimension n. In this section, we just give the statement of this generalized version of Sperner's lemma. The proof is a straightforward adaptation of the usual proof. Detailed proofs of Sperner's lemma and its generalization in pseudomanifolds can be found in many places, such as in [79], in chapter 9 of [64], or in chapter 1 of [13].

Sperner's lemma is traditionally stated in terms of *colorings* of the vertices of a simplicial complex. However, contrary to Section 1.2.1, several vertices of the same simplex can have the same color. In other words, the simplicial complex is not necessarily chromatic. When this is the case, we use the colors red, green, blue on the pictures (instead of black, gray, white) to differentiate. In this context, when a simplex has all its vertices of distinct colors, we say that it is *properly colored*. Sperner's lemma says that the number of properly-colored simplices inside a pseudomanifold is odd, provided that the colors on its boundary satisfy some properties.

The most basic statement, in two dimensions, is the following. Consider a triangle that has been subdivided into smaller triangles. We color the vertices of this subdivision according to the following rules:



- The three corner vertices are colored in red, blue and green.
- Each vertex along the subdivided boundary connecting two corner vertices is colored with one of the two colors of the corners. For example, each vertex on the red-green boundary is colored either red or green.
- The vertices in the interior of the subdivision can be given any of the three colors.

Sperner's lemma says that, for any assignment of colors that respects these rules, there will always be an odd number of properly colored triangles. In particular, there exists at least one.

To formulate the general version of this lemma, fix a finite set A of colors of cardinality $n+1$, and let $\Delta^A = (A, \mathscr{P}(A) \setminus \{\varnothing\})$ be the pure n-dimensional simplicial complex with only one facet. Note that a (non-necessarily proper) coloring of a simplicial complex $\mathcal{C}$ is the same thing as a (non-necessarily rigid) simplicial map $f : \mathcal{C} \to \Delta^A$. Moreover, we consider the *identity carrier map* $\Xi : \Delta^A \to 2^{\Delta^A}$, defined by $\Xi(X) = (X, \mathscr{P}(X))$. To each simplex $X \subseteq A$, it associates the subcomplex of Δ^A consisting of X and all of its faces.

Lemma 1.13 (Sperner's lemma)**.** *Let $\mathcal{C}$ be a pure n-dimensional simplicial complex, $f : \mathcal{C} \to \Delta^A$ a coloring of the vertices of $\mathcal{C}$, and $\Phi : \Delta^A \to 2^{\mathcal{C}}$ a carrier map, such that:*

- *for every $X \in \Delta^A$, the subcomplex $\Phi(X)$ is a pseudomanifold with boundary,*
- *for every $X \in \Delta^A$, Φ commutes with the boundary operator, i.e., $\partial\Phi(X) = \Phi(\partial X)$, and*
- *$f \circ \Phi$ is carried by Ξ,*

then $\mathcal{C}$ has an odd number of properly colored n-simplices.

Proof. Cf. [64], chapter 9. □

Intuitively, the carrier map Φ, along with the first two conditions, identifies a subcomplex of $\mathcal{C}$ that is "shaped like" Δ^A. The third condition says that f is a Sperner coloring: for instance, given an edge $X \subseteq A$ of Δ^A, the fact that $f \circ \Phi(X) \subseteq \Xi(X)$ says that all the vertices of $\Phi(X)$ must be colored using only the two colors of the edge X.

1.2.6 The Index lemma

In this section, we introduce another important theorem of combinatorial topology, called the *Index lemma*. It will only be used in Section 1.4, in order to showcase how the solvability of a task can be related to subtle topological properties. The Index lemma can be understood as a more refined version of Sperner's lemma. Indeed, Sperner's lemma, which is equivalent to Brouwer's fixed point theorem, essentially says that there is no surjective continuous map from an n-dimensional ball onto an $(n-1)$-dimensional sphere.

So, it is about the nonexistence of continuous maps whose images have holes. The Index lemma relates to a more subtle topological distinction: it is not only about the image of the map, but about how many times and in which directions the continuous map is winding around holes.

As for Sperner's lemma, this intuition is expressed combinatorially by counting the properly-colored n-simplices inside a pseudomanifold. The new notion that intervenes is that of *orientation*: each simplex can be oriented either *positively* or *negatively*. Then, the properly-colored simplices are counted *by orientation*, meaning that two simplices with opposite orientations cancel each other. More precisely, the Index lemma relates the number of properly-colored n-simplices inside a coherently oriented pseudomanifold, to the number of properly colored $(n-1)$-simplices on the boundary.

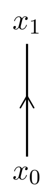
Consider a set X of $n+1$ elements. A *permutation* of X is an ordered sequence S of the elements of X. We denote as S_i the *i-th* element of the sequence S, where $0 \leq i \leq n$. A *transposition* of S consists of interchanging the position of two elements S_i, S_j of S, with $i \neq j$. A permutation S' of X is an *even transposition* of another permutation S if S' can be obtained from S by applying an even number of transpositions. An *odd transposition* of S is defined similarly. This defines a partition of the set of permutations of X into two equivalence classes, one containing the even transpositions of S (and S itself), the other containing the odd transpositions of S.

Let $\mathcal{C}$ be a pure n-dimensional simplicial complex, and $X = \{x_0, x_1, \ldots, x_n\} \in \mathcal{C}$ be an n-simplex. An *orientation* of X is a set consisting of a permutation of its vertices and all even transpositions of this permutation. If $n > 0$, there are exactly two possible orientations for X: the sequence $\langle x_0, x_1, \ldots, x_n \rangle$ and all its even permutations, and the sequence $\langle x_1, x_0, x_2, \ldots, x_n \rangle$ and all its even permutations. In dimension $n > 1$, an orientation of X *induces* an orientation on all of its $(n-1)$-faces as follows. If X is given the orientation $\langle x_0, x_1, \ldots, x_n \rangle$, and if $Y_i := X \setminus \{x_i\}$ denotes the $(n-1)$-face of X without vertex x_i, then

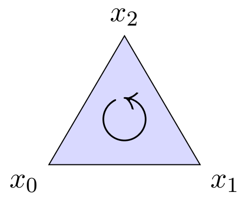
– if i is even, then Y_i gets the orientation $\langle x_0, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \rangle$,

– if i is odd, then Y_i gets the opposite orientation $\langle x_1, x_0, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \rangle$.

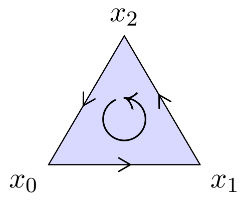
For example, in dimension 2, if $X = \{x_0, x_1, x_2\}$ is oriented $\langle x_0, x_1, x_2 \rangle$, then its faces Y_0, Y_1, and Y_2 get the orientations $\langle x_1, x_2 \rangle$, $\langle x_2, x_0 \rangle$ and $\langle x_0, x_1 \rangle$, respectively. In the picture below, oriented simplices are represented in 1 and 2 dimensions. In dimension 1, the two possible orientations correspond to the two directions of an edge. In dimension 2, the two orientations can be depicted as "clockwise" and "counterclockwise".



1-simplex
oriented $\langle x_0, x_1 \rangle$



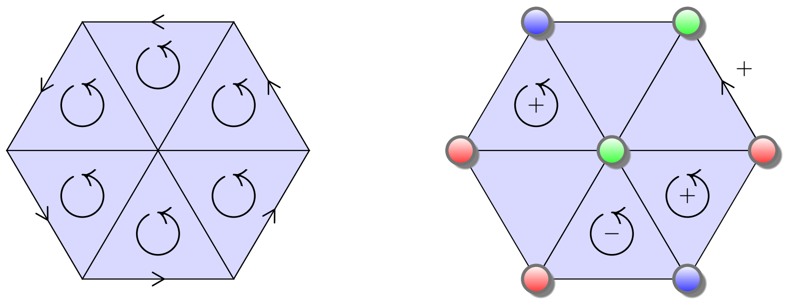
2-simplex
oriented $\langle x_0, x_1, x_2 \rangle$



induced orientations
on the faces

Definition 1.14. A *coherently oriented* pseudomanifold $\mathcal{K}$ of dimension n is a pseudomanifold $\mathcal{K}$ equipped with an orientation for each of its n-simplices such that if $X, X' \in \mathcal{K}$ share an $(n-1)$-face Y, then the two orientations on Y induced by X and X' are opposite.

A 2-dimensional coherently oriented pseudomanifold is depicted below (left), along with the induced orientation on its boundary.



We now fix a set A of $n+1$ colors. Without loss of generality, we assume that our colors are natural numbers: $A = \{0, \dots, n\}$. On the 2-dimensional pictures, we use colors red, green, blue, corresponding respectively to the values $0, 1, 2$. Let X be a k-simplex and $\chi : X \to A$ a proper coloring of the vertices of X. Let $a_0, \dots, a_k \in A$ be the colors of the vertices of X in increasing order. We say that X is *oriented positively* if the sequence of vertices $\langle x_0, \dots, x_k \rangle$, with $\chi(x_i) = a_i$, belongs to the orientation of X. Otherwise, we say that X is *oriented negatively*. Given a set of oriented properly-colored simplices, the number of simplices *counted by orientation* is the number of positively oriented simplices, minus the number of negatively oriented simplices (in other words, positive counts for $+1$, negative for -1).

In the next definition, if $i \in A$ is a color, we write $A_i := A \setminus \{i\}$.

Definition 1.15. Consider a coherently oriented pseudomanifold $\mathcal{K}$ with the induced orientation on its boundary $\partial(\mathcal{K})$. Let χ be a coloring, not necessarily proper, of the vertices of $\mathcal{K}$ with the colors A.

– The *content* of $\mathcal{K}$, written $C(\mathcal{K})$, with respect to χ is the number of the properly colored n-simplices of $\mathcal{K}$, counted by orientation.
– The *index* of $\mathcal{K}$, written $I_i(\mathcal{K})$, with respect to χ and $i \in A$, is the number of properly colored $(n-1)$-simplices of $\partial(\mathcal{K})$ with colors A_i, counted by orientation.

If there is no ambiguity, we omit $\mathcal{K}$ and write C or I_i. The next lemma is the restatement of Corollary 2 in [40] using our notations. This formulation that we use here can be found in [64, ch. 12]. See also [61, pp. 46-47] for a simple version in dimension 2.

Lemma 1.16 (Index Lemma). *Let $\mathcal{K}$ be a coherently oriented pseudomanifold of dimension n, colored with A. Then $C(\mathcal{K}) = (-1)^i I_i(\mathcal{K})$.*

Proof. Cf. [64], chapter 12. □

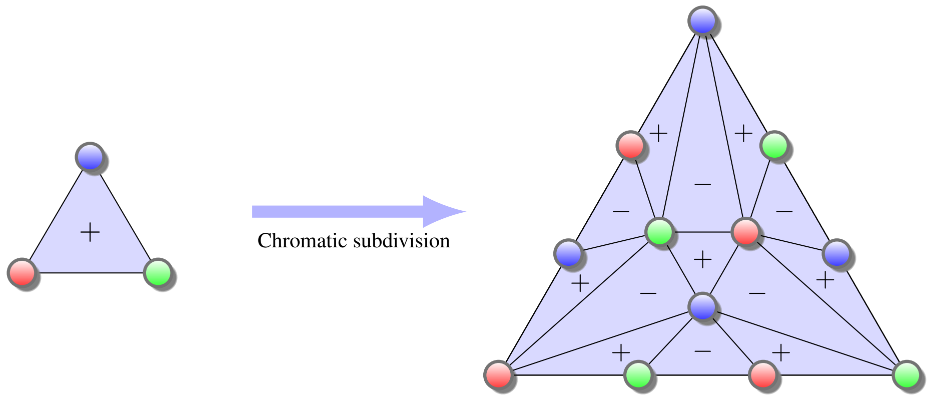
On the example above, the content is $C = +1$, and the index when removing the blue color is $I_2 = +1$. One can also check that $I_0 = +1$ and $I_1 = -1$ (recall that $0, 1, 2$ stands for red, green, blue). The coloring χ of $\mathcal{K}$ can be seen as a simplicial map $\chi : \mathcal{K} \to \Delta^A$ from $\mathcal{K}$ to the standard n-simplex Δ^A. Thus, we can think of the index of $\mathcal{K}$ as the number of times that $\partial(\mathcal{K})$ is wrapped around $\partial(\Delta^A)$, i.e., a combinatorial version of the notion of *degree* in topology.

To compute the index and content of chromatic subdivisions, we can just compute it from the original complex before subdivision happens. Indeed, we have the following Lemma:

Lemma 1.17. *Let X be a properly colored n-simplex with colors from A. Let $\mathsf{ChSub}(X)$ be a chromatic subdivision of X. Then $C(\mathsf{ChSub}(X)) = C(X)$.*

Proof. Assume w.l.o.g. that X is counted positively. Thus, $C(X) = 1$. We proceed by induction on n. For $n = 0$ (X is a single vertex), chromatic subdivisions do nothing so the result trivially holds. Now assume X is of dimension n. For clarity, we now write the dimension n of $\mathcal{K}$ as a superscript when we write the content $C^n(\mathcal{K})$ or index $I_i^n(\mathcal{K})$.

By the Index lemma, $C^n(\mathsf{ChSub}(X)) = (-1)^n\, I_n^n(\mathsf{ChSub}(X))$. Let Y_n be the $(n-1)$-face of X with colors from A_n. Since $\mathsf{ChSub}(X)$ is a chromatic subdivision of X, the subcomplex $\mathcal{K}$ induced by the properly colored $(n-1)$-simplexes on the boundary of $\mathsf{ChSub}(X)$ with colors in A_n, is a chromatic subdivision of Y_n. Thus, $I_n^n(\mathsf{ChSub}(X)) = C^{n-1}(\mathcal{K}) = C^{n-1}(Y_n)$ by induction hypothesis, and $C^{n-1}(Y_n) = I_n^n(X) = (-1)^n\, C^n(X)$. This concludes the proof. $\square$



Now, assume we start with a coherently oriented pseudomanifold $\mathcal{K}$, and apply a chromatic subdivision to each n-simplex. Every properly colored simplex will still have the same contribution to the content (even after subdivision), and each non-properly colored simplex will not contribute to the content (even after subdivision). So, the content remains unchanged, and the same goes for the index:

Corollary 1.18. *Chromatic subdivisions preserve the index and content.*

1.3 Distributed computing through combinatorial topology

We now show how we can model tasks and protocols using chromatic simplicial complexes. We work with $n+1$ processes, in order to get simplicial complexes of dimension n. Moreover, we will usually assume that the process names are natural numbers $[n] := \{0, \ldots, n\}$, and we refer to the i-th process as either “process i” or just “P_i”. The process names will also be referred to as *colors*, since we use them to color the vertices of our chromatic simplicial complexes. Intuitively, a k-dimensional simplex (for $k \leq n$) represents an execution in which $k+1$ processes are participating, and the others have crashed. The set of participating processes is the set of colors of the vertices of this simplex.

1.3.1 The task specification

A task is specified by giving two chromatic simplicial complexes $\mathcal{I}$ and $\mathcal{O}$, called respectively the *input complex* and the *output complex*, along with a carrier map from $\mathcal{I}$ to $\mathcal{O}$. Intuitively, $\mathcal{I}$ describes all the possible combinations of input values that the processes can start with, and $\mathcal{O}$ the combinations of output values that they are allowed to decide. For that purpose, we suppose given a set Val of *values* that can be exchanged between processes (usually, Val contains at least the natural numbers).

Definition 1.19 (Simplicial task)**.** A *task* is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$, where:

– $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I}, \ell_\mathcal{I})$ is a pure chromatic simplicial complex of dimension n, colored with process numbers, and equipped with a labeling $\ell_\mathcal{I} : V_\mathcal{I} \to \mathsf{Val}$ of input values, such that each vertex is uniquely determined by its color and label.

– $\mathcal{O} = (V_\mathcal{O}, S_\mathcal{O}, \chi_\mathcal{O}, \ell_\mathcal{O})$ is a pure chromatic simplicial complex of dimension n, colored with process numbers, and equipped with a labeling $\ell_\mathcal{O} : V_\mathcal{O} \to \mathsf{Val}$ of output values, such that each vertex is uniquely determined by its color and label.

– $\Delta : \mathcal{I} \to 2^\mathcal{O}$ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{O}$.

The purpose of the labelings is just to make explicit our intuition that the vertices of $\mathcal{I}$ and $\mathcal{O}$ should be pairs (i, v), where $i \in [n]$ is a process number and $v \in \mathsf{Val}$ is an input or output value. So, an n-simplex X of $\mathcal{I}$ attributes one input value to each process. The image of this simplex under the carrier map $\Delta(X)$ is a pure n-dimensional subcomplex of $\mathcal{O}$ (because Δ is rigid), corresponding to the set of acceptable outputs. But this carrier map specifies more than that: if we start with a k-dimensional face $Y \subseteq X$, meaning that only some processes participate in the computation, then $\Delta(Y)$ specifies which k-dimensional outputs are acceptable for these participating processes. The fact that Δ is color-preserving means that the process numbers do not change during the computation; and the fact that Δ is monotonic says that $\Delta(Y) \subseteq \Delta(X)$, that is, if the "crashed" processes were just being slow, they can still manage to produce a valid output afterwards.

So, this definition indeed corresponds to the intuitive idea of an input-output relation that we gave in Section 1.1.1, along with a specification of what happens when only a subset of processes participate.

Examples

Example 1.20 (Consensus). In the *consensus* task, the processes can be given as input any value in Val.

– The input complex has vertices $V_\mathcal{I} = [n] \times \mathsf{Val}$, where a vertex (i, v) is colored by i and labeled by v. The input simplexes are all the sets of vertices that are properly colored.

– The output complex has the same set of vertices, $V_\mathcal{O} = [n] \times \mathsf{Val}$, with the same coloring and labeling; but the only output simplexes are those of the form $\{(0, v), (1, v), \ldots, (n, v)\}$ for each value $v \in \mathsf{Val}$, along with their faces.

– The carrier map Δ sends an input simplex $\{(i_0, v_0), \ldots, (i_k, v_k)\}$ to all the output simplexes with the same set of participating processes, where all processes decide on the same v_i.

The *binary consensus* task is a variant of consensus where the only possible values are 0 and 1.

Example 1.21 (k-Set-agreement). *k-Set-agreement* (for $k \leq n$) is a weaker variant of consensus, where instead of agreeing on one common output, we require that the cardinality of the set of decision values should be at most k. In particular, 1-set-agreement is the same as consensus.

– The input complex $\mathcal{I}$ is the same as the one of consensus.

– The output complex $\mathcal{O}$ has the same set of vertices, but the maximal simplexes are those of the form $\{(0, d_0), (1, d_1), \ldots, (n, d_n)\}$, such that $|\{d_0, \ldots, d_n\}| \leq k$.

– The carrier map Δ sends an input simplex $\{(i_0, v_0), \ldots, (i_k, v_k)\}$ to all the output simplexes with the same set of participating processes, and where each decision value is *valid*: $d_i \in \{v_0, \ldots, v_k\}$.

Example 1.22 (Adopt-commit). The *adopt-commit* task [3] is yet another variant of consensus, which can be used to implement round-based protocols for set-agreement.

– The inputs are taken among a finite set of cardinality $m + 1$, i.e., $V_{\mathcal{I}} = [n] \times [m]$. All the properly colored input simplices are allowed.
– The processes can make two kinds of decisions, either (ADOPT, v) or (COMMIT, v), for some $v \in [m]$. The allowed output simplices are those that satisfy the *coherence* condition: if one process decides on (COMMIT, v), then every process decides either on (COMMIT, v) or (ADOPT, v).
– The carrier map Δ formalizes the two following conditions:
    - *validity:* every decision value is among the inputs of the participating processes,
    - *convergence:* if all the inputs are v, then all the outputs are (COMMIT, v).

Example 1.23 (Validity). *Validity* is an even weaker version of consensus and set-agreement, where we only keep the requirement that the decision values should be valid, i.e., among the input values of the processes. There is no agreement requirement at all.

– $\mathcal{I}$ is the same as for consensus.
– All output simplexes are allowed: $\mathcal{O} = \mathcal{I}$.
– The carrier map Δ sends an input simplex $\{(i_0, v_0), \dots, (i_k, v_k)\}$ to all the output simplexes with the same set of participating processes, and where each decision value is *valid*: $d_i \in \{v_0, \dots, v_k\}$.

Example 1.24 (Weak symmetry breaking). The *weak symmetry breaking* task is interesting in the context of *anonymous* protocols, where the processes do not have access to their PID. As input, each process has a name $p \in \Pi$, where usually there are many more possible names than processes $|\Pi| \gg n + 1$. As output, they must return a binary value 0 or 1, such that not all processes decide on the same output.

– $\mathcal{I}$ has all vertices $V_{\mathcal{I}} = [n] \times \Pi$, but not all input simplices are allowed: the processes must be given distinct names. So, an input simplex is of the form $\{(0, p_0), \dots, (n, p_n)\}$ where the p_i are distinct.
– $\mathcal{O}$ has vertices $V_{\mathcal{O}} = [n] \times \{0, 1\}$, and it contains all the properly colored simplexes, except the one where the decisions are all 0, and the one where the decisions are all 1.
– The carrier map Δ takes each k-simplex of $\mathcal{I}$ to all the simplexes of $\mathcal{O}$ with the same set of participating processes. Thus, if less than $n + 1$ processes participate, there is no restriction at all on the outputs; if $n + 1$ processes participate, the only restriction is that the two missing output simplexes are forbidden.

A related task is *renaming*, which has the same input complex, but as output the processes are required to decide names from a smaller set of names, so that all the participating processes pick distinct names.

1.3.2 The protocol complex

In the context of distributed computability, the only purpose of a protocol is to solve a task. So, the processes start the computation with input values taken from the input complex $\mathcal{I}$ of the task that we want to solve. It is helpful to separate the computation into two phases: first, the processes communicate using the shared objects at their disposal, until they reach a final state where they are ready to decide on an output value; then, when all the participating processes are decided, the decision values are returned. The simplicial definition of a *protocol* (Definition 1.25 below) corresponds to the first phase. The decision phase will be taken into account in Definition 1.29.

At the end of an execution, each process has a *local view* of the computation, which intuitively represents the partial information that it has managed to acquire during this execution. From the point of view of one process, two different executions might produce the same local view, in which case these executions are *indistinguishable* for this process. The output value that is decided by a process can only depend on its local view; so, in two indistinguishable executions, a process must decide on the same output. In the next definition, let Views denote an arbitrary set of local views.

Definition 1.25 (Simplicial protocol)**.** A *protocol* is a triple $(\mathcal{I}, \mathcal{P}, \Psi)$, where:

- $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I}, \ell_\mathcal{I})$ is a pure chromatic simplicial complex of dimension n, colored with process numbers, and equipped with a labeling $\ell_\mathcal{I} : V_\mathcal{I} \to \mathsf{Val}$ of input values, such that each vertex is uniquely determined by its color and label.
- $\mathcal{P} = (V_\mathcal{P}, S_\mathcal{P}, \chi_\mathcal{P}, \ell_\mathcal{P})$ is a pure chromatic simplicial complex of dimension n, colored with process numbers, and equipped with a labeling $\ell_\mathcal{P} : V_\mathcal{P} \to \mathsf{Views}$ of local views, such that each vertex is uniquely determined by its color and label. $\mathcal{P}$ is called the *protocol complex*.
- $\Psi : \mathcal{I} \to 2^\mathcal{P}$ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{P}$.

This is very similar to the definition of a task, except that we do not label vertices with output values but with views. Intuitively, a n-simplex of $\mathcal{P}$ represents one possible execution of the protocol. A single vertex of $v \in V_\mathcal{P}$ represents a single process, whose process number is $i := \chi_\mathcal{P}(v)$, along with its local view $\ell_\mathcal{P}(v)$. This vertex might belong to several n-simplices of $\mathcal{P}$; they correspond to all the executions that are indistinguishable for process i. Given an input simplex $X \in \mathcal{I}$, the carrier map Ψ keeps track of which final configurations are reachable starting from the input configuration X.

Examples

The most important protocol in distributed computability is called the *immediate-snapshot* protocol. It was introduced by Borowsky and Gafni [15], and has been thoroughly studied for two important reasons:

1. it is equivalent to the usual setting of read/write registers in terms of wait-free task solvability [16];
2. it preserves the topology of the input complex.

Example 1.26 (The immediate-snapshot protocol). Given any input complex $\mathcal{I}$, the *immediate-snapshot* protocol is defined as follows.

- The protocol complex is $\mathcal{P} := \mathsf{ChSub}(\mathcal{I})$, the chromatic subdivision of $\mathcal{I}$ (see Section 1.2.3).
- The carrier map $\Psi : \mathcal{I} \to 2^\mathcal{P}$ is also the one of Section 1.2.3.

Of course, this abstract specification of the protocol deserves an explanation of the computational intuition behind it. In the immediate snapshot protocol, each process has a designated memory cell where it can atomically write its input value. After doing so, it can atomically take a snapshot of the whole shared memory, in order to read the values that have been written by the other processes. Moreover, we restrict to a subset of all the executions described above: the snapshot is guaranteed to happen "immediately" after the write, in the following sense. Formally, an immediate snapshot execution consists of a sequence of sets of processes, where each set is called a *concurrency class*. All the processes in the same concurrency class will execute their write operations together. Then, they all execute their snapshots together. Then, we move on to the next concurrency class. Thus, all the processes inside a given concurrency class will see each other's values, as well as the values of the previous concurrency classes.

The *view* of a process in such an execution is the vector that was returned by the snapshot operation. In other words, it contains the set of values that it was able to observe from the other processes, along with which value belongs to which process. So, formally, a view is a set of pairs $(i, v) \in [n] \times \mathsf{Val}$ where each process appears at most once, i.e., it is a simplex of the input complex. Moreover, a process necessarily sees its own value, so a vertex of the protocol complex $\mathcal{P}$ should intuitively be a pair (i, X_i) consisting of a process number i and the local view of this process. This looks almost like the vertices of $\mathsf{ChSub}(\mathcal{I})$, except that in Definition 1.9, the first component was an i-colored vertex $v \in V_{\mathcal{I}}$. But this vertex v can be recovered since it belongs to X_i, so this distinction is irrelevant. Finally, the two conditions of Definition 1.9 correspond to the "immediacy" of the snapshot.

See [4] for a more thorough account of the immediate-snapshot model, and a formal proof that its protocol complex is indeed the standard chromatic subdivision.

Example 1.27 (The k-round immediate-snapshot protocol). The previous example was for one round of immediate-shapshot communication. This protocol can be iterated as follows. At each round, a fresh shared memory array is used. Each process writes its local view, and then it takes a snapshot of the whole array in order to obtain its next view, and so on. After repeating this action k times, the protocol complex that we obtain is an iterated chromatic subdivision.

– The protocol complex is $\mathcal{P} := \mathsf{ChSub}^k(\mathcal{I})$.

– The carrier map $\Psi : \mathcal{I} \to 2^{\mathcal{P}}$ sends each input simplex to its subdivided image.

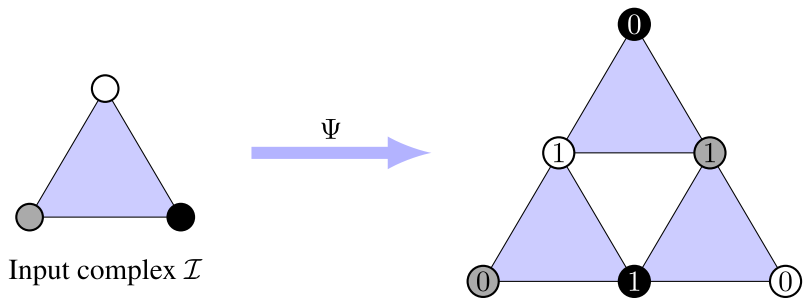
Example 1.28 (Test-and-set protocol with one flag). As a toy example, we present a very simple protocol using a single test-and-set object. A test-and-set object is a shared memory cell that contains a single bit of information. Initially, the value is set to 0. The processes can access this cell by calling its `test-and-set()` method, which takes no argument. Its effect is to atomically perform the two following operations: first, read the value of the cell, then, write the value 1. The method returns the value that has been read.

So, when n processes call this object concurrently, exactly one of them ("the winner") sees value 0, and the others ("the losers") see value 1. The *view* of a process in this execution is simply the binary digit that it reads.

– Since the processes cannot exchange values using this protocol, their input values are irrelevant, and we will consider that $\mathcal{I}$ consists of just one n-simplex: $V_{\mathcal{I}} = [n]$, and $S_{\mathcal{P}} = \mathscr{P}([n])$.

– The protocol complex $\mathcal{P}$ has vertices $V_{\mathcal{P}} = [n] \times \{0, 1\}$, colored by the first component and labeled by the second, as usual. An n-simplex of $\mathcal{P}$ is of the form $\{(0, b_0), (1, b_1), \dots, (n, b_n)\}$ where exactly one of the b_i's is 0 and the others are 1. $S_{\mathcal{P}}$ contains all the faces of these simplices.

– The carrier map $\Psi : \mathcal{I} \to 2^{\mathcal{P}}$ takes each input k-simplex (i.e., a set of $k+1$ participating processes), to the simplices of $\mathcal{P}$ of the form $\{(i_0, b_0), (i_1, b_1), \dots, (i_k, b_k)\}$, where exactly one of the b_i's is 0, along with their faces.



Input complex $\mathcal{I}$

Protocol complex $\mathcal{P}$

The picture above depicts the corresponding protocol complex for three processes, whose names are represented as colors black, gray and white. As we can see on the picture, the topology of the input complex is not preserved: a hole has been created.

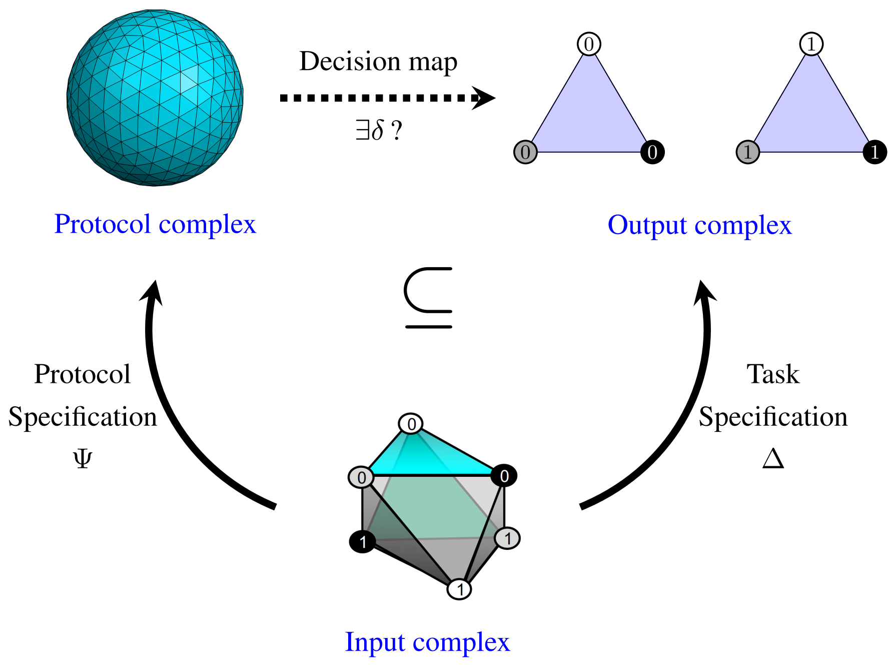
1.3.3 Definition of solvability

Now that we have defined tasks and protocols, we can define what it means for a protocol to solve a task.

Definition 1.29. A protocol $(\mathcal{I}, \mathcal{P}, \Psi)$ *solves* a task $(\mathcal{I}, \mathcal{O}, \Delta)$ if there exists a chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$, called the *decision map*, such that $\delta \circ \Psi$ is carried by Δ. That is, for all $X \in \mathcal{I}$, we must have

$$\delta(\Psi(X)) \subseteq \Delta(X)$$

The intuition behind this definition is simple: the decision map δ associates with a local view of process i an output value for this process. So, a process must decide on its output only according to its local view. The fact that $\delta \circ \Psi$ is carried by Δ says that these decisions are made according to the task specification.

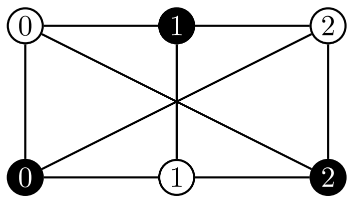


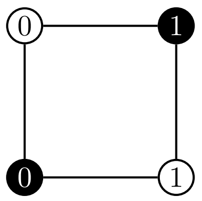
This topological characterization of task solvability is particularly interesting when we want to prove impossibility results. Indeed, to prove that a task is not solvable, one must prove the non-existence of a particular simplicial map between simplicial complexes. Many powerful mathematical tools can be used to prove such results: arguments based on k-connectedness, homology, or combinatorial theorems such as Sperner's lemma and the Index lemma.

1.4 A case study: solvability of Equality Negation tasks

In order to showcase how combinatorial topology can be used to prove impossibility results in distributed computing, we now present a study of the solvability of a class of tasks, called *equality negation tasks*, in the usual model of wait-free processes communicating through read/write registers. Unlike the previous sections of this chapter, which were concerned with introducing classic definitions from the literature, this chapter presents a new contribution of this thesis. The work presented here was done in collaboration with Éric Goubault, Marijana Lazić and Sergio Rajsbaum, and was accepted for publication at DISC 2019 [50].

The equality negation task for two processes was originally introduced by Lo and Hadzilacos [88], as the central idea to prove that the consensus hierarchy [63, 77] is not robust. Consider two processes P_0 and P_1, each of which has a private initial value, drawn from the set of possible input values $I = \{0, 1, 2\}$. Each process must decide on a binary output value, either 0 or 1, so that the decisions of the processes are the same if and only if the initial values of the processes are different. The input and output complexes of this task are depicted below.


Input complex $\mathcal{I}$


Output complex $\mathcal{O}$

The carrier map $\Delta : \mathcal{I} \to 2^{\mathcal{O}}$ takes each of the three vertical input edges (with equal inputs), to the two horizontal output edges (with different outputs); and each of the six other input edges to the two vertical output edges. As we will discuss later, there is no restriction imposed on the "solo executions", because we will consider a model without crashes.

It is proved in [88] that this task is strictly weaker than consensus, i.e., that consensus is not solvable using equality negation objects. However, like consensus, the equality negation task is not solvable using read/write registers. In [88], this fact is proved by reduction to the consensus task. They show that, under the assumption that equality negation is solvable using registers, then there would be a correct implementation of consensus using equality negation. Thus, consensus would be solvable with read/write registers, which we know is impossible.

A topological proof. In fact, there is a simple topological argument to show that there is no solution to the equality negation task in the immediate-snapshot model (which is equivalent to read/write registers).

Assume for contradiction that there is a solution, with some number of rounds, k. Then, the protocol complex $\mathcal{P}$ for k-round immediate-snapshot is a subdivision $\mathsf{ChSub}^k(\mathcal{I})$ of the input complex, where each edge is divided into 3^k edges. The carrier map $\Psi : \mathcal{I} \to 2^{\mathcal{P}}$, carries each input edge to its subdivision in the protocol complex, and each input vertex (i, v) to the i-colored extremity of that subdivided edge. Also, Definition 1.29 states that $\mathcal{P}$ solves equality negation if there exists a simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Psi$ is carried by Δ.

Assume that this is the case, and consider the subcomplex $\mathcal{C}$ of $\mathcal{I}$ induced by all edges that have distinct input values. Notice that $\mathcal{C}$ is connected (it is in fact a circle). Thus, $\Psi(\mathcal{C})$ is also connected, since it is a subdivision of $\mathcal{C}$. Since the image of a connected simplicial complex under a simplicial map is connected, $\delta(\Psi(\mathcal{C}))$ is a connected subcomplex of $\mathcal{O}$.

The specification of the equality negation task states that the decisions should be equal, for all the simplexes of $\mathcal{C}$. But the subcomplex of $\mathcal{O}$ of edges with the same decision values is disconnected. Therefore, $\delta(\Psi(\mathcal{C}))$ must be equal to one of the edges in this subgraph. Without loss of generality, we assume that $\delta(\Psi(\mathcal{C}))$ is the edge of $\mathcal{O}$ where both processes decide 0. It follows, in particular, that $\delta(\Psi(P_0,2)) = (P_0,0)$ and $\delta(\Psi(P_1,2)) = (P_1,0)$, since those two input vertices belong to $\mathcal{C}$.

Now, consider the input edge $e := \{(P_0,2),(P_1,2)\}$, where both processes have the same input. For the same reason as before, $\delta(\Psi(e))$ must be connected. Thus, $\delta(\Psi(e))$ is equal to one of the two edges of $\mathcal{O}$ with distinct output values. Without loss of generality, suppose it is the edge $\{(P_0,0),(P_1,1)\}$. This implies that $\delta(\Psi(P_1,2)) = (P_1,1)$. But this contradicts the equality $\delta(\Psi(P_1,2)) = (P_1,0)$ above. □

Equality negation tasks for n processes. The equality negation task for two processes defined in [88] does not have a unique generalization to the case with more than two processes.

In the rest of this section, we introduce a generalization that allows any number of input values, but we keep the set of possible decision values as $\{0,1\}$. It seems natural to require the following conditions:

(i) if all the processes have (pairwise) different input values, they decide on the same output, and

(ii) if they all initially have the same input value, they do not decide on the same output.

Still, this definition does not cover all possible combinations of input values of all processes, for example, if there are two processes with the same input, and a third process with a different one. Therefore, there is a lot of freedom for defining output constraints for the remaining input cases. Our goal is to analyse all the possible generalizations of a certain form.

We define a family of equality negation tasks, based on the cardinality of the set of input values of all processes. Let n be the number of processes and let there exist exactly n input values, namely $I = \{0,1,2,\dots,n-1\}$. This assumption is done w.l.o.g., as we explain at the end of the section: all our results can be extended to $|I| > n$ in a straightforward manner. Let $\mathcal{I}$ denote the corresponding input complex, where every possible combination of input values from I is allowed. In a given input simplex $X \in \mathcal{I}$, the set of input values of all processes is written $\mathit{InputSet}(X)$. As this set is always a non-empty subset of the set I of all possible input values, its cardinality ranges from 1 to n. Note that the case when all processes have the same input value is represented by $|\mathit{InputSet}(X)| = 1$, and the case when all processes have different input values is the one when $|\mathit{InputSet}(X)| = n$. Different constraints on the intermediate cases introduce a family of tasks.

For every $k, \ell \in \mathbb{N}$ with $1 \leq k < \ell \leq n$, we define a generalized equality negation task as follows:

- If we have $|\mathit{InputSet}(X)| \geq \ell$, then all processes must decide on the same value, either 0 or 1.
- If $|\mathit{InputSet}(X)| \leq k$, then not all processes decide on the same output value, i.e., there must be at least one process deciding 0, and at least one deciding 1.
- If $k < |\mathit{InputSet}(X)| < \ell$, then each process can decide independently on any value from $\{0,1\}$.

We propose to study the solvability of this task, depending on the values of the two parameters k and ℓ. The original equality negation task for two processes of Lo and Hadzilacos [88] is recovered as a special case, where $n = 2$, $I = \{0,1,2\}$, $k = 1$ and $\ell = 2$. Note that the same task with $I = \{0,1\}$ is trivially solvable; this is a peculiarity of the two-process case. We discuss this fact at the end of the section.

The computational model. We study the solvability of this task in the usual immediate-snapshot model (see Example 1.26). However, there is one little oddity: we do not consider crashing processes. All the n processes are present in the beginning of an execution, and they are guaranteed to run until the end of their program. This is a simplifying assumption: there is no need to wonder about what the specification of a solo execution should be (how does one single process manage to "disagree"?). Instead of crashes, the two ingredients that allow us to obtain impossibility results are asynchrony and wait-freedom. Although a bit unusual, this assumption is not restrictive: if we wanted to transpose our results to a model where processes can crash, we could simply say that the task specification does not impose any restrictions on the outputs whenever at least one process has crashed.

1.4.1 Solvable cases

We claim that whenever there is a “gap” between the two parameters k and ℓ, where the processes are allowed to decide on any output value without constraint, the equality negation task is solvable.

```
1  Integer v := input_value
2
3  $P_0$: V := immediateSnapshot(v);
4      return 0;
5
6  $P_i$: V := immediateSnapshot(v);
7      if |V| $\geq 2$ and |val(V)| $\leq 1$ then
8        return 1
9      else return 0;
```

Figure 1.1: Case $k = 1$ and $\ell = n$, for $n \geq 3$.

```
1  Integer v := input_value
2
3  $P_0$: V := immediateSnapshot(v);
4      return 0;
5
6  $P_i$: V := immediateSnapshot(v);
7      if |V| $\geq$ n+k$-\ell$+1 and |val(V)| $\leq$ k then
8        return 1
9      else return 0;
```

Figure 1.2: Arbitrary k and ℓ with $\ell - k \geq 2$.

For simplicity, we first focus on the special case when $n \geq 3$, $k = 1$ and $\ell = n$. A protocol to solve this task is presented in Figure 1.1. First, notice that the processes are not *anonymous*: they can execute a different piece of code depending on their process number. Here, the program run by process P_0 is indicated at lines 3-4; whereas all the other processes P_i for $1 \leq i \leq n$ run the program at lines 6-9. In this figure, the input value of a the process is named `v`. The operation `immediateSnapshot(v)` returns the vector `V` of values that were observed by the snapshot. `|V|` denotes the number of processes whose values appear in `V`, and `val(V)` denotes the set of values that appear in `V`. In this algorithm, process P_0 participates in the immediate-snapshot, and then it decides 0 independently of its input and the result of the snapshot. All the other processes decide on their output value according to the following rule: if P_i sees at least 2 processes (including itself), and their input values are the same, then P_i decides 1. Otherwise, it decides 0.

Proposition 1.30. *The algorithm of Figure 1.1 solves equality negation if $n \geq 3$, $k = 1$ and $\ell = n$.*

Proof. As $k = 1$ and $\ell = n$, we need to show that the following two conditions are satisfied: (i) when all processes have the same input value, they will disagree, and (ii) when all processes have different inputs, then they all decide on the same value (in this case, 0).

(i) The distinguished process P_0 always decides 0, so we need to make sure that at least one process will decide 1. Among the last two processes that execute their snapshots, one is not P_0. This process, say P_j, will see at least $n - 1$ processes. Since $n - 1 \geq 2$, and since all processes have the same input value, P_j decides 1, which is enough for the disagreement.

(ii) Suppose that no two processes have the same input value. Then all the processes in this will agree and decide 0. Indeed, the only way that a process can decide 1 is when it has seen 2 processes with the same input value. But that would contradict our assumption. □

In the general case, for any n, k and ℓ such that $\ell - k \geq 2$, the idea is the same. The corresponding algorithm is given in Figure 1.2. As before, process P_0 always decides 0. All the other processes decide according to the following rule: if it sees at least $n + k - \ell + 1$ processes, and among their input values there are at most k different values, the process decides 1, and otherwise it decides 0.

Theorem 1.31. *The algorithm of Figure 1.2 solves the equality-negation task if $\ell - k \geq 2$.*

Proof. The two conditions that we want to prove are the following. For every input configuration X, (i) if $|InputSet(X)| \leq k$, then at least one process decides 0 and at least one decides 1; and (ii) if $|InputSet(X)| \geq \ell$, then all processes decide on the same value, here 0.

(i) Again, we have process P_0 that decides 0 and there is a process P_j, with $j \neq 0$, that sees at least $n - 1$ processes. Since $\ell - k \geq 2$ (by assumption), we have that $n - 1 \geq n + k - \ell + 1$, and thus P_j will indeed decide 1.

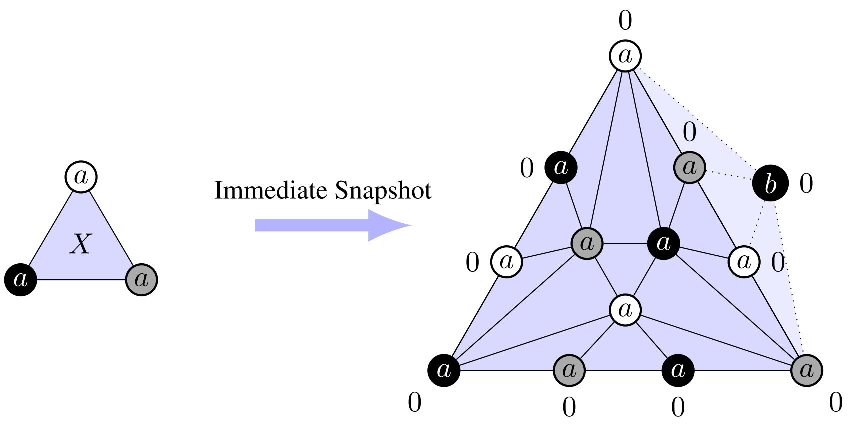
(ii) By means of contradiction, suppose that a process P_i decides 1. Then P_i has seen at least $n+k-\ell+1$ processes with at most k different input values. Note that P_i did not see any process with the remaining $\ell - k$ values, and that there must be at least one process with each of those values. So the total number of processes would be at least $n + 1$ processes, which is not the case. ☐

Thus, we have proved that all variants of the equality negation task are solvable whenever $k + 2 \leq \ell$. Since by definition the two parameters k and ℓ are such that $1 \leq k < \ell \leq n$, the only remaining cases are those with $\ell = k + 1$. In the following, we discuss the solvability of the tasks depending on the value of k. The other parameter ℓ is now always assumed to be $k + 1$.

1.4.2 Impossibility proof when k is small

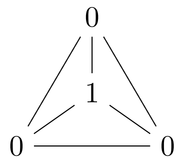
For simplicity, we first look at the case where $k = 1$; we will see later that it can be easily generalized to any $k \leq n/2$. For $k = 1$, the goal of the task is the following. If all the input values are the same, then the processes should disagree (i.e., not all of them should decide on the same output). In all other cases, the processes should agree.

Let us assume for contradiction that the task is solvable in the immediate snapshot model. We focus on what happens in one of the initial configurations where all the processes have the same input value a. Let us call $X = \{(P_0, a), \ldots, (P_{n-1}, a)\}$ the corresponding simplex of the input complex. After the processes exchange information using iterated immediate snapshot communications, we obtain in the protocol complex a chromatic subdivision $\mathsf{ChSub}^k(X)$ of the original simplex X. The situation for 3 processes and one round of immediate snapshot is depicted below. The color of a vertex represents the process name; the value written inside a vertex is its input value; and next to it is the decision value.



In the input complex, the simplex X is surrounded by other simplices where not all inputs are the same. Thus, after the immediate snapshot computation occurs, the boundary of the subdivided simplex $\mathsf{ChSub}^k(X)$ is still surrounded by simplices with a different input value (three of them are depicted above, with input value b). In these simplices, the task specifies that all the processes must decide on the same output. Assume w.l.o.g. that this output is 0. Since the boundary of $\mathsf{ChSub}^k(X)$ is connected, we can propagate the 0's step by step and we obtain that every vertex on the boundary of $\mathsf{ChSub}^k(X)$ must decide on value 0.

To reach a contradiction, we will show that, given any assignment of output values 0 or 1 to the inner vertices of $\mathsf{ChSub}^k(X)$, there will always be at least one simplex of $\mathsf{ChSub}^k(X)$ where all the outputs are equal. This contradicts the task specification. Note that the next part of the proof only works because $\mathsf{ChSub}^k(X)$ is a *chromatic* subdivision; the statement does not hold for other subdivisions, as depicted on the right. If we regard the output values 0 and 1 as colors, the result that we want to prove looks like Sperner's lemma (Section 1.2.5), where instead of counting the properly colored simplices inside the subdivision, we want to count the monochromatic ones, with respect to output values 0 and 1. To be able to use Sperner's lemma in this situation, we use a recoloring trick which was already used in [21] to obtain lower bounds on the renaming problem.



Suppose we are given an assignment of output values to the vertices of $\mathsf{ChSub}^k(X)$, such that all the vertices on the boundary decide 0. For v a vertex of $\mathsf{ChSub}^k(X)$, we write $d_v \in \{0,1\}$ the decision value of v, and $\mathsf{id}_v \in \{0,\dots,n-1\}$ its process number. We define the recoloring of v to be $c_v := (\mathsf{id}_v + d_v) \mod n$. We have the following property:

Lemma 1.32. *A simplex S of $\mathsf{ChSub}^k(X)$ is monochromatic w.r.t. the decision values d_v if and only if it is proper w.r.t. the recoloring c_v.*

Proof. Since $\mathsf{ChSub}^k(X)$ is a chromatic subdivision of the simplex X, every simplex S of $\mathsf{ChSub}^k(X)$ is properly colored w.r.t. the process numbers id_v. Thus, if S is monochromatic w.r.t. the decision values d_v (that is, if all its vertices have the same decision value d), it is clear that the recoloring $c_v = (id_v + d) \mod n$ will still be proper. Conversely, we proceed by contraposition: suppose that not all decision values are the same. Then, there must be two vertices v and w of S, such that $\mathsf{id}_w = (id_v + 1) \mod n$ and $d_v = 1$ and $d_w = 0$. Thus $c_v = c_w$, and the recoloring c is not proper. □

We can now easily conclude the proof. Since on the boundary of $\mathsf{ChSub}^k(X)$ all the vertices have the decision value 0, the recoloring c is just the process number: $c_v = \mathsf{id}_v$ for every boundary vertex v. Moreover, since $\mathsf{ChSub}^k(X)$ is the (iterated) chromatic subdivision of a single input simplex X, we know that the process numbers id_v form a Sperner coloring on its boundary. Thus, by Sperner's lemma, there must exist a simplex $S \in \mathsf{ChSub}^k(X)$ that is properly colored w.r.t. the recoloring c, and by Lemma 1.32 this means that all the vertices of this simplex decide on the same output. Therefore, the equality negation task with parameter $k = 1$ cannot be solved by iterated immediate snapshot.

The same proof can be adapted for any $k \leq n/2$:

Theorem 1.33. *The equality negation task for $n \geq 3$ processes, with parameters $k \leq n/2$ and $\ell = k+1$ is not solvable in the immediate snapshot model.*

Proof. The only difference with the case $k = 1$ above is that we now start with the simplex $X = \{(P_0, 0), \ldots, (P_{k-1}, k-1), (P_k, 0), \ldots, (P_{2k-1}, k-1), \ldots\}$, which has exactly k distinct input values, and every input appears at least twice. Therefore, if we remove one vertex from X, we obtain a face of X (of cardinality $n-1$), which still has k distinct inputs. Then, this face belongs to another simplex with $k+1$ input values. Thus, after subdivision, every process on the boundary of X must decide on the same output (say, 0). Now the rest of the proof is exactly the same as in the case of $k = 1$: we have a simplicial complex $\mathsf{ChSub}^k(X)$ which is a chromatic subdivision of X, and we know that every process on its boundary has decision value 0. By Sperner's lemma, there is a simplex $S \in \mathsf{ChSub}^k(X)$ that is properly colored w.r.t. the recoloring c, and by Lemma 1.32 it is monochromatic w.r.t. the decision values. Since X has $\leq k$ distinct input values, this set of outputs is not allowed. $\square$

1.4.3 Impossibility proof when $\mathrm{n} - \mathrm{k}$ is odd

We now extend the proof of Section 1.4.2 to show that the task is not solvable by iterated immediate snapshot in half of the remaining cases: namely, whenever $n - k$ is odd, the task is unsolvable. We will rely on the same recoloring trick (and the associated Lemma 1.32), but instead of Sperner's lemma, we will use a more refined topological property, the Index lemma (see Section 1.2.6).

Low-dimensional example

Before we proceed to the general construction, let us illustrate the idea of the proof using the particular case of 3 processes and parameter $k = 2$. This instance of the task is the following: if the three input values are different, then the processes should decide on the same output; otherwise, they should disagree. The input complex $\mathcal{I}$ contains every possible assignment of input values $\{0, 1, 2\}$ to the three processes $\{P, Q, R\}$. More formally, its maximal simplices are of the form $X_{ijk} = \{(P, i), (Q, j), (R, k)\}$, for all $i, j, k \in \{0, 1, 2\}$. We can decompose this input complex $\mathcal{I}$ into two parts (depicted in Figure 1.3 below):

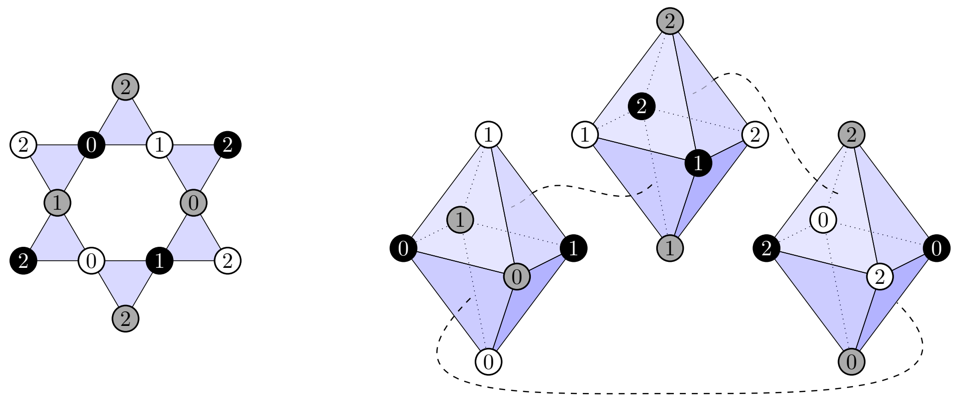


Figure 1.3: Exploded view of the input complex for $n = 3$ processes. All vertices with the same color and input value should be identified.

- There are 6 maximal simplices X_{ijk} such that $|\{i, j, k\}| = 3$, that is, where all three processes have distinct values. In each of these simplices, the processes should agree on a common output value; moreover, since this subcomplex is connected, this common output has to be the same in all six of them.

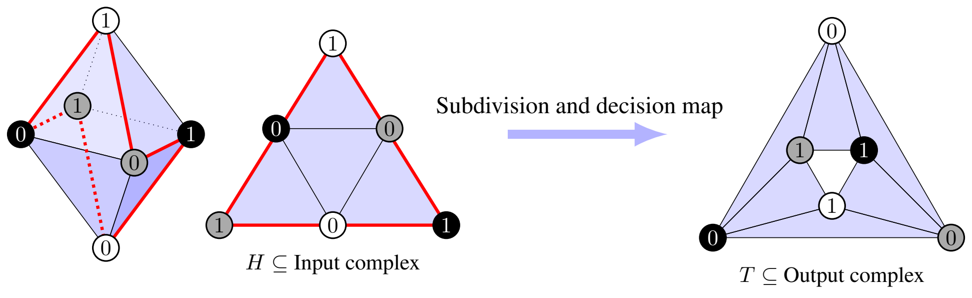
– The rest of the input complex consists of simplices X_{ijk} such that $|\{i,j,k\}| \leq 2$. In that part of the input, the processes should not agree. This part of the input complex is topologically a "pearl necklace" of three spheres.

Note that the 6 simplices with three distinct inputs are actually filling the hole in the middle of the "necklace". Thus, $\mathcal{I}$ is homotopy equivalent to a wedge of 2-spheres; in the terminology of [64], it is a *pseudosphere*.

We now focus on one of the three spheres depicted in Figure 1.3; for example, the one where all inputs are either 0 or 1. That sphere has six edges that are labeled with two distinct input values 0 and 1. Each of these 01-edges also belongs to one of the six simplices with three inputs $0, 1, 2$: therefore, on each of these edges, all processes have to decide on the same output value (say, w.l.o.g., that the common output value is 0). In the picture below, the 01-edges of the sphere are depicted in red. They form a circle on the surface of the sphere, splitting it in two half-spheres. We now look at only one of those two half-spheres, and call it H. It consists of four 2-simplices, and its boundary contains only 01-edges. Thus, after the immediate snapshot computation occurs, this complex will be subdivided, and in order to solve the task, we should satisfy the following two conditions:

(1) All the vertices on the (red) boundary must be mapped to the same output, say, 0; and
(2) In every triangle, not all processes should decide on the same output.

This means we should have a simplicial map from a chromatic subdivision of H to the subcomplex T of the output complex where not all decision values are the same.


Subdivision and decision map
$H \subseteq$ Input complex
$T \subseteq$ Output complex

Since the processes on the boundary of H all decide on output 0, the boundary of H has to be mapped to the outside boundary of T. In this situation, it seems clear topologically that some simplex inside H will necessarily be sent to the "111-hole" in the middle of T, contradicting condition (2). Notice however that the boundary of H is winding *twice* around the boundary of T. Moreover, this winding number is preserved by the chromatic subdivisions of H induced by the immediate snapshot protocol. Sperner's lemma, that we used in Section 1.4.2, only works if the boundaries are matched exactly (i.e., if we have a Sperner coloring). That's why we need to use the index lemma, which is able to handle cases where one boundary is winding several times around the other.

Proposition 1.34. *The equality negation task for $n = 3$ processes, with parameters $k = 2$ and $\ell = 3$ is not solvable in the immediate snapshot model.*

Proof. Assume for contradiction that the task is solvable, and let H be the subcomplex of the input complex described above. So, there should be a chromatic subdivision $\mathsf{ChSub}^k(H)$ of H, and an assignment of decision values to its vertices, which solves the task. Remember that all the vertices on the boundary of $\mathsf{ChSub}^k(H)$ must decide on the same output value, for example 0. We choose an arbitrary (coherent) orientation for the simplices of H; and we assign colors using the same recoloring as in Section 1.4.2. What we want to do now is prove that the content of $\mathsf{ChSub}^k(H)$ is non-zero. Using the index lemma, we can also compute its index. And by Corollary 1.18, applied to the boundary of $\mathsf{ChSub}^k(H)$, it is equal to the index of H itself.

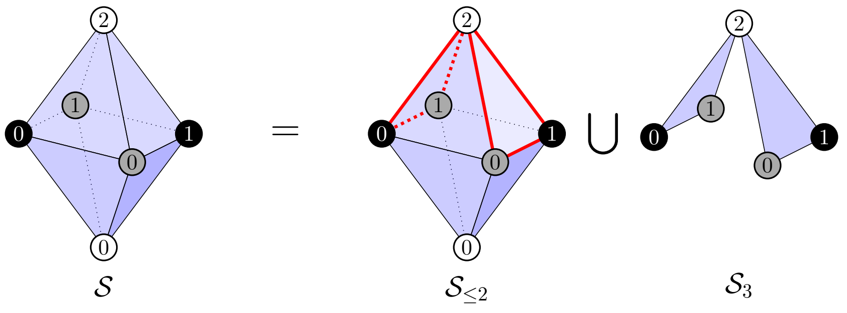
Then, an easy calculation shows that the index of H is either 2 or -2 depending on the orientation that we chose. So, no matter how many rounds of immediate snapshot we do, the index of $\mathsf{ChSub}^k(H)$ is either 2 or -2. By the index lemma, its content must be equal to its index (in absolute value). Since it is non-zero, there must exist proper triangles in $\mathsf{ChSub}^k(H)$, which by Lemma 1.32 correspond to monochromatic triangles w.r.t. the decision values. This contradicts the solvability of the task. $\square$

General case

We now work with any k and n (remember however that $k < n$ since $\ell = k+1 \leq n$), and we try to follow the same recipe as in the previous section. So, our goal is to find a subcomplex of the input complex, which is a pseudomanifold (in order to be able to apply the index lemma), and whose boundary consists of simplices with exactly k distinct input values (so that we can say that every process on the boundary has to decide on the same output). A simple way to obtain a pseudomanifold is to choose a sphere in the input complex, and restrict ourselves to a subcomplex of this sphere. Although this idea is inspired from what we did in the low-dimensional example, the reader should be warned that the general construction we are about to do, when instantiated with $n = 3$ and $k = 2$, does not give the same proof as in Proposition 1.34.

Consider the subcomplex $\mathcal{S}$ of the input complex, which contains all the vertices of the form (P_i, a_i), where $a_i = 0$ or $a_i = i$ if $1 \leq i \leq k$, and $a_i = 0$ or $a_i = 1$ for all other values of i. The simplices of $\mathcal{S}$ are all combinations of such vertices, which are properly colored w.r.t. the process names. For example, $X = \{(P_0, 0), (P_1, 0), \ldots, (P_{n-1}, 0)\}$ is a simplex of $\mathcal{S}$. For ease of notation, when talking about maximal simplices, we omit the process names and just write $X = \langle 0, \ldots, 0 \rangle$, where the i-th component is the value of process P_i. Another simplex of $\mathcal{S}$ is $\langle 0, 1, 2, \ldots, k, 0, \ldots, 0 \rangle$.

For the case $n = 3$, $k = 2$, the sphere $\mathcal{S} = \{\langle a, b, c \rangle \mid a, b \in \{0, 1\}, c \in \{0, 2\}\}$ is represented below (the colors black, gray, white correspond to P_0, P_1, P_2, respectively).



Since every process can take exactly two different input values, independently from the other processes, the complex $\mathcal{S}$ is a $(n-1)$-sphere. In particular, it is a pseudomanifold. Among the maximal simplices of $\mathcal{S}$, some of them contain exactly $k+1$ distinct input values, and the others contain k values or fewer. We write $\mathcal{S}_{k+1} \subseteq \mathcal{S}$ for the subcomplex of $\mathcal{S}$ that contains all the simplices with $k+1$ input values, and $\mathcal{S}_{\leq k} = \mathcal{S} \setminus \mathcal{S}_{k+1}$. Both $\mathcal{S}_{k+1}$ and $\mathcal{S}_{\leq k}$ are pseudomanifolds, and they have the same boundary $\partial \mathcal{S}_{k+1} = \partial \mathcal{S}_{\leq k} = \mathcal{S}_{k+1} \cap \mathcal{S}_{\leq k}$. In every simplex of $\mathcal{S}_{k+1}$, the processes must decide on the same value (since $\mathcal{S}_{k+1}$ is connected); assume w.l.o.g. that they decide on the output value 0.

The complex $\mathcal{S}_{\leq k}$ will play the role of the complex H that we had in Section 1.4.3. In the case of $n = 3$, $k = 2$, it corresponds to the sphere $\mathcal{S}$ with two holes corresponding to the two simplices with three distinct inputs. Its boundary is represented in red in the picture above. So the situation is a bit different than what we had in Section 1.4.3: instead of one boundary winding twice around the output, we now have two holes, each of them winding once around the output complex. Fortunately, the index lemma can also deal with such cases, as long as the two holes are winding in the same direction around the output (otherwise they would cancel each other). This is taken into account when we compute the index: here, one can check that the two holes have the same orientation, so the index will be either 2 or -2. For general k and n, the boundary of $\mathcal{S}_{\leq k}$ can have many holes, each of them winding several times around the output complex. Our goal is once again to prove that the index is non-zero.

Theorem 1.35. *The equality negation task for $n \geq 3$ processes, with parameters k and $\ell = k + 1$ such that $n - k$ is odd, is not solvable in the immediate snapshot model.*

Proof. After the immediate snapshot communication occurs, we obtain a chromatic subdivision $\mathsf{ChSub}^k(\mathcal{S}_{\leq k})$. We already know that it is a pseudomanifold, and that on its boundary every process has to decide 0. Our goal is now to use the index lemma to prove that there exists a simplex inside $\mathsf{ChSub}^k(\mathcal{S}_{\leq k})$ which is monochromatic w.r.t. the decision values. First, we use the recoloring of Lemma 1.32, so that we are now searching for a properly colored simplex in $\mathsf{ChSub}^k(\mathcal{S}_{\leq k})$. All we need to show is that the content of $\mathsf{ChSub}^k(\mathcal{S}_{\leq k})$ is non-zero; thus, we want to compute its index. Since the index is preserved by chromatic subdivisions (see Corollary 1.18), we just need to compute the index of $\mathcal{S}_{\leq k}$. Since the boundary of $\mathcal{S}_{\leq k}$ is the same as the boundary of $\mathcal{S}_{k+1}$, their index is the same (in absolute value), and since the index of $\mathcal{S}_{k+1}$ is equal to its content, we want to compute the content of $\mathcal{S}_{k+1}$. In $\mathcal{S}_{k+1}$, every simplex is properly colored (because everyone decides 0), so we just need to count the simplices of $\mathcal{S}_{k+1}$ by orientation. If the result is non-zero, this concludes the proof.

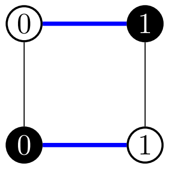
So let us pick an arbitrary orientation, say that the simplex $\langle 0, \dots, 0 \rangle$ is oriented positively. Each time we change the value of one coordinate, the orientation changes: for example, the simplex $\langle 1, 0, \dots, 0 \rangle$ is oriented negatively. Let us first characterize the maximal simplices of $\mathcal{S}_{k+1}$: they are of the form $\langle a_0, a_1, 2, 3, \dots, k, a_{k+1}, \dots, a_{n-1} \rangle$, where for each $i \notin \{2, \dots, k\}$, $a_i \in \{0, 1\}$, and at least one of the a_i must be 0, and at least one must be 1. There are exactly $2^{n-k+1} - 2$ such simplices: all combinations of 0's and 1's are possible, except for $z = \langle 0, 0, 2, 3, \dots, k, 0, \dots, 0 \rangle$ and $u = \langle 1, 1, 2, 3, \dots, k, 1, \dots, 1 \rangle$. To go from z to u, we need to change $n - k + 1$ coordinates. Thus, since $n - k + 1$ is even, then z and u have the same orientation; otherwise, they would have opposite orientations.

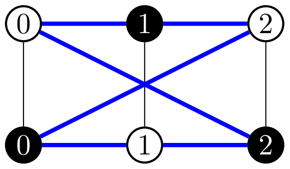
A simple calculation shows that summing the orientations of all the simplices of $\mathcal{S}_{k+1}$, plus z and u, gives 0 as a result. Indeed, if we omit the middle components "$2, 3, \dots, k$", these 2^{n-k+1} simplices correspond to all the binary sequences of size $n - k + 1$. We can, for example, enumerate those sequences using Gray code, so that the orientations are alternating, hence there is the same number of positively and negatively oriented simplices. Since z and u are not in $\mathcal{S}_{k+1}$, and they have the same orientation, then the content of $\mathcal{S}_{k+1}$ must be either 2 or -2. In any case, it is non-zero: so the task is not solvable whenever $n - k$ is odd. □

In particular, the case of $k = n - 1$ is not solvable for any n, as $n - k = 1$ is odd. Note that in this proof, we do not really use the full power of the index lemma: we use it to show that the content of $\mathcal{S}_{\leq k}$ is equal (in absolute value) to the content of $\mathcal{S}_{k+1}$. This also follows from the simple fact that the content of a sphere is always 0, which is a direct consequence of the index lemma; so the contents of $\mathcal{S}_{\leq k}$ and $\mathcal{S}_{k+1}$ must be opposite.

1.4.4 Discussion on the number of input values

Since the beginning of this section, we have been working with a set of input values $I = \{0, \dots, n-1\}$ of size n, for a number $n \geq 3$ of processes. This might seem a bit surprising, considering that in the original equality negation task for two processes [88], the size of the input set matters a lot. For $I = \{0, 1\}$, the task is solvable, but for $I = \{0, 1, 2\}$, it becomes unsolvable. It is quite informative to think about why our unsolvability proofs of Theorems 1.33 and 1.35 fail in the case of two processes and $I = \{0, 1\}$. Both of them fail for a similar reason. In those proofs, we identify a subcomplex of the input complex ($\mathsf{ChSub}^k(X)$ in Section 1.4.2 and $\mathcal{S}_{\leq k}$ in Section 1.4.3), and we surround its boundary by simplexes where every process must decide on the same output. However, in order to assume that every vertex on the boudary decides on the same output 0, we need to know that the part of the input complex which decides on the same output is connected. For two processes, this is not the case if $I = \{0, 1\}$, however taking $I = \{0, 1, 2\}$ ensures connectedness. Both input complexes are depicted below; the subcomplex where decisions should be the same is represented in blue.





No such problem occurs when there are more than 3 processes, as the relevant subcomplexes remain connected regardless of the number of input values. Nevertheless, we can still wonder what happens when we allow an input set I of size $|I| > n$. Actually, all the results of this section can be extended to such a setting in a straightforward way:

- Theorem 1.31 can easily be shown to work when more than n input values are allowed: intuitively, the algorithm relies only on the size of the sets of values that are seen by the processes. It never looks at the actual values.
- In all our unsolvability proofs (Theorems 1.33 and 1.35, and Proposition 1.34), we proceed by picking a subcomplex of the input complex, and then we work in this subcomplex to find a contradiction. If we add more input values, this just makes the input complex bigger, but all those proofs still work as they stand.

Conclusion. For some values of the parameters (when $k > n/2$, $\ell = k + 1$ and $n - k$ is even), the solvability of the equality negation tasks remains open. We conjecture that they should be unsolvable. The same proof method as in Section 1.4.3 using the Index lemma might work for the remaining cases, but we would need to find another subcomplex of the input complex on which to apply it. There are a number of other variations of this task that we could have studied. For example, instead of having binary outputs in $\{0, 1\}$, we could have a larger set of output values. Then, we could introduce two more parameters to define what it means to “agree” or to “disagree” in that context. Such parameters appear in the generalized symmetry breaking task [20].

CHAPTER 2

Trace semantics

The typical framework in which one studies distributed computing is that of asynchronous processes communicating through shared objects. A wide variety of computational models have been introduced, depending on the communication primitives the processes are allowed to use, and the assumptions made on the kind of failures that might happen during the computation. The area of fault-tolerant computability [64] studies what kind of *decision tasks* can be solved in such models. To solve a task, each process starts with a private input value, and after communicating with the other processes, it has to decide on an output. For instance, a well-known task is *consensus*, where the processes must agree on one of their inputs.

A very well-studied setting is the one of wait-free protocols communicating through shared read/write registers. In order to prove impossibility results in this context, people started developing powerful mathematical tools based on algebraic topology [14, 112]. The fundamental paper by Herlihy and Shavit [70] provides a topological characterization of the tasks that can be solved by a wait-free protocol using read/write registers. First, they showed that the *specification* of a task gives rise to a *carrier map* between two chromatic simplicial complexes called the *input complex* and the *output complex*. The *Asynchronous Computability Theorem*, which says that a read/write protocol solves a decision task if and only if there exists a map from a subdivision of the input complex to the output complex that is *carried* by the task specification. The subdivided simplicial complex that appears in this theorem was the first occurrence of a *protocol complex*, a very compact way of expressing combinatorially the partial knowledge that each process acquires during the computation.

Soon thereafter, it became clear that this method actually extends beyond the setting of read/write registers: many other computational models can also be described as protocol complexes. This idea gave birth to the *combinatorial topology* approach to distributed computing, as described in Section 1.3 and in [64]. Generalizing the ideas behind the asynchronous computability theorem, one can define an abstract notion of protocol using *carrier maps* between an input complex and a protocol complex. Then, we can take Herlihy and Shavit's characterization as the *definition* of what it means for a protocol to solve a task. This abstract approach is very appealing because it offers a great deal of generality: for example, the set-agreement task cannot be solved in any computational model whose protocol complex is a pseudomanifold [64, Chapter 9]. We are thus able to prove impossibility results for a wide class of computational models, instead of studying them one at a time. In principle, most computational models and synchronization primitives can be formulated using the protocol complex formalism: it has been used in the literature to model processes communicating using test-and-set objects [66], (N, k)-consensus objects [65], weak symmetry breaking and renaming [44], or various synchronous or asynchronous message-passing primitives [68, 69].

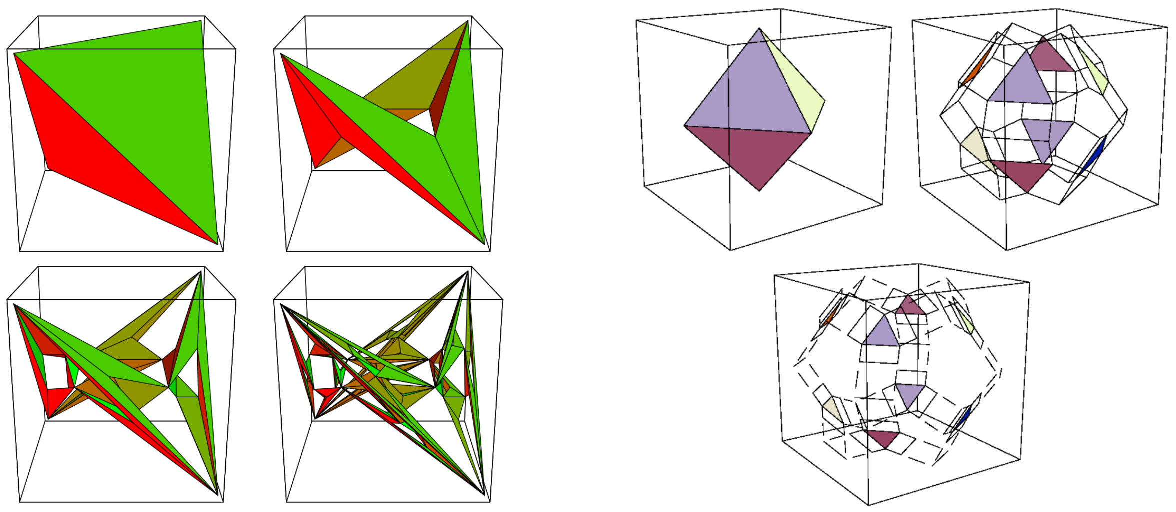


Figure 2.1: Protocol complexes for test-and-set [65] (left) and synchronous message-passing [69] (right).

The drawback of this high level of abstraction is that we have to *trust* that our protocol complex faithfully represents the behavior of the objects that we want to model. Concretely, when we use this approach to prove that some task is unsolvable in some computational model, what we really prove is just that there is no simplicial map between some simplicial complexes having some particular properties. In the case of wait-free read/write register computation, Herlihy and Shavit's asynchronous computability theorem provide an additional guarantee: namely, that the existence of such a simplicial map corresponds to a more basic notion of solvability. But for other objects such as test-and-set, there is no such guarantee. When we want to define, for example, the protocol complex which is supposed to model test-and-set computation, we must define it carefully so that the simplicial notion of "solving" a task will agree with the concrete operational semantics of test-and-set objects. Ideally, one would have to prove another version of the asynchronous computability theorem for test-and-set protocols, relating a concrete notion of solvability with the abstract simplicial definition. This is usually not done in practice, since it is often intuitively clear that the proposed notion of protocol complex makes sense operationally.

In this chapter, our main goal will be to fix this gap by generalizing the Asynchronous Computability Theorem to a large class of concurrent objects. To achieve this goal, there are three main technical steps.

(1) Define a notion of *concurrent object* which is as general as possible. In particular, it should be general enough to include all the examples from the literature that we mentioned above.
(2) Define an operational semantics for concurrent processes communicating through arbitrary shared objects. The goal is to obtain a concrete definition of *solving* a task.
(3) Derive the protocol complex from the operational semantics, and prove a *generalized asynchronous computability theorem*, saying that the simplicial notion of solvability agrees with the concrete one.

By proving this theorem, we increase our trust in the topological approach to task solvability. But this is not the only benefit of this work. Indeed, along the way, we will give a precise definition of how the protocol complex should be defined, for arbitrary objects. In the literature, there are many informal descriptions of how the protocol complex can be constructed in practical examples, by putting together the "local views" of the processes in every possible computation. Depending on the context, the notion of "view" can be defined as a partial history of the computation, or some local state of the process. Since the definition of protocol complex that we give in this chapter is proved to be correct (with respect to our operational semantics), it can be instantiated in practical examples as a point of comparison, to make sure that some other ad-hoc definition is also correct.

Finally, another point of interest of this chapter will be to understand the protocol complex approach in terms of *compositionality*. By definition, the protocol complex is a huge, monolithic combinatorial structure, which contains information about all the possible executions of the whole protocol. However, in computer science, a very important notion to study programs is *modularity*: a complex system is built and analyzed by assembling basic blocks to construct a more complex one, and so on. So, what happens to the protocol complex when we *compose* programs? Can we combine two protocol complexes to create a larger one? Conversely, can we decompose a large protocol complex into smaller components, in order to understand it better? We do not give definitive answers to these questions in this chapter, but we will keep them in mind when defining our operational semantics.

Related work. Of course, the original paper of Herlihy and Shavit [70] is our main point of comparison, since it contains the Asynchronous Computability Theorem that we intend to generalize. Another notable generalization of this theorem was proved by Saraph, Herlihy and Gafni [113], where they give a topological characterization of task solvability using *t-resilient* read/write protocols, instead of wait-free. Our approach is somewhat orthogonal, since we keep the wait-free assumption, but instead work on generalizing the class of objects that the processes are allowed to use.

The notion of concurrent object specification that we use is closely related to Lamport's notion of specification [85, 84]. In order to discuss the relevance of our notion, we compare it to other specification techniques for concurrent objects, namely, linearizability [71] and interval-linearizability [18]. In fact, the formalism that we use is largely inspired by those linearizability-based techniques. We came up with the so-called *expansion property* that we impose on our concurrent specifications when trying to understand interval-linearizability, and indeed, thanks to it, the class of concurrent objects that we are able to specify exactly corresponds to the interval-linearizable objects. The trace-based operational semantics that we provide is slightly unusual, since the processes are allowed to use non-linearizable objects. However, other similar models can be found in the literature, e.g. in [41]. It is also related to I/O automata [91] in the sense that every object call is decomposed into an invocation and a response, which can be interleaved arbitrarily. Yet, as far as we know, our computational model is the only one that completely abstracts the notion of *internal state* of the objects, and relies exclusively on their high-level specification.

Plan. We first start with a simple approach in Section 2.1, where we prove an asynchronous computability theorem which is limited to processes communicating through linearizable objects. Since this case is easier and more intuitive than the more general setting that we use afterwards, it will serve as an outline for the rest of the chapter. The Sections 2.1.1, 2.1.3 and 2.1.4 correspond to the three main steps of the proof (1), (2), (3) that we described in the introduction.

Then, the main contribution of the chapter is given in the next three sections, once again following the same outline. In Section 2.2, we define a very general way of specifying concurrent objects. We argue the relevance of this definition in Section 2.2.3, where we prove Theorem 2.32 which relates nicely several specification techniques. In Section 2.3, we define a concrete computational model, which uses the concurrent specifications of the previous section. Finally, in Section 2.4, we define the protocol complex and prove our *generalized asynchronous computability theorem* in Theorem 2.73. Along the way, we also study the relationship between tasks and one-shot objects, in Theorem 2.66. The work presented in these three sections has been published in two papers, [52] for Sections 2.2 and 2.3, and [86] for Section 2.4.

The last Section 2.5 relates work in progress about the compositionality of the protocol complex. Its main purpose is to reformulate the computational model of Section 2.3 in the language of *game semantics*.

2.1 A first approach

This section describes our first attempt at defining the protocol complexes associated with general objects; and a generalized version of the asynchronous computability theorem. The reason why this is just a "first attempt" is not that it fails: in the end, we *do* obtain a satisfying definition of protocol complex, which allows us to formulate an extended version of the asynchronous computability theorem. But the notion of object specification that we use here is too restrictive. It includes basic low-level primitives like read/write registers, test-and-set, compare-and-swap, as well as linearizable data structures such as lists or trees. This contrasts with the recent discovery [18] that many of the objects that are used in fault-tolerant computability are of a more complex nature. Indeed, they are usually not linearizable, meaning that each individual call to such an object can perform many consecutive atomic operations, each of which can be observed individually by the other processes. In contrast, a linearizable object must behave as if each method call is atomic.

Even if the end result is not completely satisfactory, this first approach is intuitive and easy to follow. It will serve as an outline of what we want to do in the rest of this chapter, once we have a more powerful notion of object specification. The approach that we take here is based on a paper by Goubault, Mimram and Tasson [55], where the immediate-snapshot object (called *scan/update* in their paper) is studied. They compare three different kinds of semantics: a simple trace-based semantics; the protocol complex approach; and the so-called directed space semantics. Here, we focus on the trace semantics, which we extend to a slightly more general setting. The directed space semantics will be the topic of Chapter 4. All the definitions and notations that we introduce here are local to this section. They will be redefined in Sections 2.2, 2.3 and 2.4, in a more general setting.

2.1.1 Objects specifications

We fix a number n of processes and a set $\mathcal{V}$ of values that contains some distinguished value $\perp$, representing an uninitialized value. If $\bar{v} \in \mathcal{V}^n$ is a tuple of values, we write $v_i \in \mathcal{V}$ for the i-th component of $\bar{v}$, and $\bar{v}[i \leftarrow x] \in \mathcal{V}^n$ for the vector $\bar{v}$ where the i-th component has been replaced by x. We write $[n]$ for the set $\{0, \dots, n-1\}$, whose elements are called *process numbers*. In our setting, the n processes start the computation together, and each of them is given a private input value. In order to communicate, they have access to shared *objects*: for example, they might be able to write and read a shared memory, or to synchronize using test-and-set objects. Each process has its own program, and they run in an asynchronous way, meaning that any interleaving of the actions of each process can occur. At the end of the computation, each process must decide individually on an output value.

Our first step is to define the notion of shared object. It can be thought of as a black box, with several *methods* that a process can call in order to obtain a response. To avoid working with too many parameters, we will assume that the methods do not have arguments. This is done without loss of generality: for instance, with a read/write register, the two method calls `write(1)` and `write(2)` will be modeled by to two distinct methods m and m'.

Definition 2.1 (Object specification)**.** An *object* is specified by a set $\mathcal{M}$ of *methods*, along with

- a set of memory states Q,
- an initial state $q_{\text{init}} \in Q$,
- a transition function $\delta : Q \times \mathcal{M} \times [n] \to Q$,
- a return function $\text{ret} : Q \times \mathcal{M} \times [n] \to \mathcal{V}$.

Intuitively, when process i calls the method m from the memory state q, the object goes to the new memory state $\delta(q, m, i)$, and it returns the value $\text{ret}(q, m, i)$ to the calling process.

Example 2.2 (Write-snapshot object). This is usually described as an object with two methods, `write` and `snapshot`. We suppose given a shared memory array of size n. The method `write(v)` performed by process i writes the value `v` in the i-th memory cell. The method `snapshot()` performed by process i reads the whole memory array atomically, and returns it. For the moment, we do not impose any restriction on how these two methods are used; the processes are not obligated to alternate between writes and snapshots, there is no "immediate snapshot" condition, and the processes can write any value (the protocol might not be "full information").
As explained above, the `write` method is split into many methods w^x for each value $x \in \mathcal{V}$:

- The set of methods is $\mathcal{M} = \{s\} \cup \{w^x \mid x \in \mathcal{V}\}$.
- The set of memory states is $Q = \mathcal{V}^n$.
- The initial state is $q_{\text{init}} = (\bot, \dots, \bot) \in \mathcal{V}^n$.
- Taking a snapshot does not change the memory state: $\delta(q, s, i) = q$.
  Snapshots return (an encoding of) the memory state: $\text{ret}(q, s, i) = \ulcorner q \urcorner$.
- Writing puts the value x in the i-th memory location: $\delta(q, w^x, i) = q[i \leftarrow x]$.
  Writing does not return a value: $\text{ret}(q, w^x, i) = \bot$.

Note that the ret function of the snapshot method should return the memory array $q \in \mathcal{V}^n$, but our definition says that ret returns a single value. We simply assume that there is a suitable encoding $\ulcorner - \urcorner$ of $\mathcal{V}^n$ into $\mathcal{V}$, and we do not worry too much about this detail in the future.

Example 2.3 (One test-and-set flag). A *test-and-set flag* is a single shared memory cell, containing a binary value 0 or 1. In the beginning of the computation, it contains the value 0. This object has a single method, `test-and-set()`, which atomically performs the two following operations: read the old value of the cell, and set it to 1. The old value of the flag is returned to the calling process. So, when several processes use this object concurrently, only one of them (the first to perform the `test-and-set()` method) will see the value 0, and all the others will see the value 1. This object is specified as follows:

- There is only one method, $\mathcal{M} = \{t\}$.
- The set of memory states is $Q = \{0, 1\}$.
- The initial state is $q_{\text{init}} = 0$.
- Always set the flag to 1: $\delta(q, t, i) = 1$.
  Return the old value of the flag: $\text{ret}(q, t, i) = q$.

Example 2.4 (Several test-and-set flags + read/write registers). When a test-and-set flag has been used, it cannot be reset to 0. So, one might want to consider several such flags. Also, since test-and-set flags do not allow the processes to exchange their input values, we might want to add shared read/write registers. Here is an object combining a test-and-set flags (each with one method `test-and-set()`), and b read/write registers (each with two methods `read()` and `write(v)`).

- The set of methods is $\mathcal{M} = \{t^k \mid k \in [a]\} \cup \{r^\ell \mid \ell \in [b]\} \cup \{w^{\ell,x} \mid \ell \in [b], x \in \mathcal{V}\}$.
- The set of memory states is $Q = \{0,1\}^a \times \mathcal{V}^b$
- The initial state is $q_{\text{init}} = (0^a, \bot^b)$
- Set the k-th flag to 1: $\delta((q,q'), t^k, i) = (q[k \leftarrow 1], q')$.
  Return the old value of the k-th flag: $\text{ret}((q,q'), t^k, i) = q_k$.
- Reading a register does not change the state: $\delta((q,q'), r^\ell, i) = (q,q')$.
  Reading the ℓ-th register returns its value: $\text{ret}((q,q'), r^\ell, i) = q'_\ell$.
- Write the value x in the ℓ-th register: $\delta((q,q'), w^{\ell,x}, i) = (q, q'[\ell \leftarrow x])$.
  Writing a register does not return a value: $\text{ret}((q,q'), w^{\ell,x}, i) = \bot$.

From now on, we suppose fixed an object specification $\langle \mathcal{M}, Q, q_{\text{init}}, \delta, \text{ret}\rangle$. A pair $(m,i) \in \mathcal{M} \times [n]$ consisting of a process number and a method is called an *action*, and is written m_i for short (since we never consider tuples of methods, there should be no confusion with the notation q_i denoting the i-th component of a tuple q). Given a process number i, we write $\mathcal{M}_i = \{m_i \mid m \in \mathcal{M}\}$ for the set of actions of process i. Thus, the set of actions is $\mathcal{A} = \bigcup_{i\in[n]} \mathcal{M}_i = \mathcal{M} \times [n]$. Finally, we write $\mathcal{T} = \mathcal{A}^*$, for the free monoid on $\mathcal{A}$, called the *trace monoid*. Elements of $\mathcal{T}$ are called *execution traces* or just *traces*, written with capital letters T, T'. Thus, a trace is just a finite sequence of actions. Since the transition function $\delta : Q \times \mathcal{A} \to Q$ is defined on all the basic elements of $\mathcal{T}$, we can extend it to a right monoid action $\delta : Q \times \mathcal{T} \to Q$ as follows:

$$\delta(q, \varepsilon) := q$$
$$\delta(q, a \cdot T) := \delta(\delta(q,a), T)$$

This just means that an execution trace is executed from left to right, as expected. For instance, taking the write-snapshot object of Example 2.2, with 4 processes, a possible execution trace is $T = w_1^1 \cdot w_0^0 \cdot s_3 \cdot w_3^{42}$. Executing it from the initial state $q_{\text{init}} = (\bot,\bot,\bot,\bot)$, we obtain the state $\delta(q_{\text{init}}, T) = (0,1,\bot,42)$. Moreover, the value returned by the snapshot is $\ulcorner(0,1,\bot,\bot)\urcorner$ since it occurs before the last write.

Remark 2.5. Depending on the object that we consider, two execution traces might be *equivalent*, in the sense that they have the same effect on the memory states Q of the object. More precisely, $T \sim T'$ when for every $q \in Q$, $\delta(q,T) = \delta(q,T')$. In general, we are only interested in studying traces up to equivalence. In [55], this equivalence relation (for the write-snapshot object) is thoroughly studied. It is shown to have a topological interpretation in the directed space setting, as it corresponds to dihomotopy equivalence of directed paths. Here, for simplicity, we will not worry about this equivalence and simply study traces individually.

2.1.2 Tasks

Once we have fixed one or several objects that our processes are allowed to use in order to compute, the goal of distributed computability is to determine which *tasks* can be solved using these objects. A task for n processes is simply a relation between n-tuples of input values, and n-tuples of output values. Let $\mathcal{I} \subseteq \mathcal{V}$ be a set of *input values*, and $\mathcal{O} \subseteq \mathcal{V}$ be a set of *output values*, such that $\bot \in \mathcal{I} \cap \mathcal{O}$.

Definition 2.6. A *task* Θ is a relation $\Theta \subseteq \mathcal{I}^n \times \mathcal{O}^n$ such that for all $(\bar{v}, \bar{v}') \in \Theta$ and $i \in [n]$,

– $v_i = \bot$ if and only if $v'_i = \bot$.
– there is $\bar{v}'' \in \mathcal{O}^n$ such that $(\bar{v}, \bar{v}'') \in \Theta$ and $(\bar{v}[i \leftarrow \bot], \bar{v}''[i \leftarrow \bot]) \in \Theta$.

The *domain* of Θ, written dom $\Theta \subseteq \mathcal{I}^n$, is the set of all input assignments $\bar{v} \in \mathcal{I}^n$ such that there exists $\bar{v}' \in \mathcal{O}^n$ such that $(\bar{v}, \bar{v}') \in \Theta$. When trying to solve such a task Θ, the n processes will start the computation with some combination input values $\bar{v} \in \mathcal{I}^n$. At the end of the computation, each process will individually decide on an output value, thus yielding a tuple of output values $\bar{v}' \in \mathcal{O}^n$. The goal of the processes is to collectively decide their outputs so that $(\bar{v}, \bar{v}') \in \Theta$. Since we also want to model computations where some processes might crash, the symbol $\bot$ indicates that a process does not participate in a computation.

The first condition asserts that the processes that do not participate in the computation do not have a specified behavior; and the processes that do participate are always required to output some value. The second condition asserts that for every possible combination $\bar{v}$ of input values, there is some valid execution that still remains valid if process i does not participate. This reflects the idea that a very slow process cannot be distinguished from a dead process: if process i takes too long to start, all the other processes will eventually behave as if process i did not participate in the computation.

Remark 2.7. Definition 2.6 is a direct reformulation of the usual simplicial simplicial definition of tasks (as in Definition 1.19). This presentation with tuples is perhaps less elegant than the one with simplicial complexes and carrier maps, but the point here is to give a concrete definition of what it means to solve a task, without relying on topological notions. We will discuss tasks much more in depth in Section 2.2.1 and in Section 2.4.1; the above definition is just a placeholder in order to make this “first approach” section self-contained.

2.1.3 Protocols

Suppose fixed an object $\langle \mathcal{M}, Q, q_{\text{init}}, \delta, \text{ret} \rangle$ as in Definition 2.1 (note that, as in Example 2.4, it can actually comprise several objects), and a task $\Theta \subseteq \mathcal{I}^n \times \mathcal{O}^n$. We now want to define how several processes can use this object to communicate in order to solve a task. A *protocol* consists of several processes, each process having its own program and local memory. The processes communicate by calling the methods $m \in \mathcal{M}$ of the shared object(s). In the next definition, $\mathcal{M} \sqcup \mathcal{O}$ denotes the disjoint union of $\mathcal{M}$ and $\mathcal{O}$.

Definition 2.8. A *program* (for process i) is given by
– A set of local states L_i and an injection $\iota_i : \mathcal{I} \to L_i$.
– A decision function $d_i : L_i \to \mathcal{M} \sqcup \mathcal{O}$ such that $d_i(\iota_i(\bot)) = \bot$.
– A transition function $\tau_i : L_i \times \mathcal{V} \to L_i$.

Intuitively, at the beginning of the computation, the the process is given an input value $v \in \mathcal{I}$ and goes to local state $\iota_i(v) \in L_i$. The decision function says what is the next step taken by the process: either call a method, or decide on an output value. If a method is called, it returns a value in $\mathcal{V}$. The transition function says what the new local state will be, depending on the value returned by the method.

Remark 2.9. We could also have defined a programming language, with a syntax and semantics. Here, for simplicity, we use instead this automata-like definition, which roughly corresponds to the control-flow graph of the program. It allows us to forget about all the insignificant details such as local computations, and go directly from one method call to the next one.

Definition 2.10. A *protocol* $\langle L_i, d_i, \tau_i \rangle_{i\in[n]}$ consists of a program for each process.

Example 2.11 (Test-and-set protocol for consensus). We use the object of Example 2.4, with one test-and-set flag `t` and two read-write registers `r1` and `r2`. The pseudo-code below describes a protocol for two processes, solving consensus using these three objects.

```
$P_0$:  consensus(v) {
1         r$_0$.write(v);
2         x := t.test-and-set();
3         if (x == 0)
4           return v;
5         else
6           v' := r$_1$.read();
7           return v';
8       }
```

```
$P_1$:  consensus(v) {
1         r$_1$.write(v);
2         x := t.test-and-set();
3         if (x == 0)
4           return v;
5         else
6           v' := r$_0$.read();
7           return v';
8       }
```

We can easily translate the pseudo-code of P_0 to an automaton as in Definition 2.8. Recall that the set of methods $\mathcal{M}$ is the one of Example 2.4. The local states L_0 will store the values of the local variables `v`, `v'` and `x` used by the program, as well as the current position of the program counter. To be fully formal, we can take $L_0 = \mathcal{I} \times \mathcal{I} \times \{\perp, 0, 1\} \times \{1, 2, 4, 6, 7\}$, where the last component indicates the current line in the program, with the injection $\iota_0(v) = (v, \perp, \perp, 1)$.

At first, process P_0 is in some state $(v, \perp, \perp, 1) \in \mathcal{L}_0$ corresponding to its input value $v \in \mathcal{I}$. Its first action is to write v in the register r_0, so the decision function is $d_0((v, \perp, \perp, 1)) = w^{0,v} \in \mathcal{M}$. This write operation has only one possible return value $\perp$, so after the write is performed, the automaton will go to the state $\tau_0(q_{\text{init}}, \perp) = (v, \perp, \perp, 2) \in L_0$, indicating that we are at the second line of the program, with input value v, and the other two local variables are still undefined.

From this new state (let us call it q), the next action is to call the test-and-set flag, so $d_0(q) = t \in \mathcal{M}$. This call will return a value, either 0 or 1, which is stored in the local variable `x`; and depending on this value the next action is either at line 4 or line 6 of the program. So, the transition function τ_0 can go into two new states: either $\tau_0(q, 0) = (v, \perp, 0, 4) \in L_0$ (let us call it q') if the test-and-set returns 0, or $\tau_0(q, 1) = (v, \perp, 1, 6) \in L_0$ (let us call it q'') if the test-and-set returns 1.

In state q', we are at line 4 of the program, and the process has decided to output the value v. So, we have $d_0(q') = v \in \mathcal{O}$, and the value of the transition function for this state does not matter. In state q'' (line 6), the next action is to read the register r_1, so $d_0(q'') = r^1 \in \mathcal{M}$. This read operation returns a value v', which we want to remember, so the transition function $\tau_0(q'', v') = (v, v', 1, 7)$ will go to many different states, one for each value of v'. From these new states, the decision function will be $d_0(q_{v'}) = v' \in \mathcal{O}$, since the next action is to return the value v'.

The *global state* of a protocol is an element $(\ell, q) \in (\prod_i L_i) \times Q$. We write $\ell_i \in L_i$ for the local state of process i. The q component is the memory state of the object (from Definition 2.1). Recall from Section 2.1.1 that the transition function δ of the object can be extended to a right monoid action $\delta : Q \times \mathcal{T} \to Q$. We can further extend it to act on global states instead of just memory states, and hence we get $\Delta : ((\prod_i L_i) \times Q) \times \mathcal{T} \to (\prod_i L_i) \times Q$, defined as follows:

$$\begin{aligned}
\Delta((\ell, q), \varepsilon) &= (\ell, q) \\
\Delta((\ell, q), m_i) &= (\ell[i \leftarrow \tau_i(\ell_i, \mathrm{ret}(q, m, i))], \delta(q, m, i)) \\
\Delta((\ell, q), m_i \cdot T) &= \Delta(\Delta((\ell, q), m_i), T)
\end{aligned}$$

Note that in the above definition, we have not used the decision function d_i of the programs. The action of Δ is defined on every trace, even those where the sequence of actions has no connection with the "code" of the program. Thus, we need to define the *valid* traces, i.e., traces where each action is taken according to the program of the corresponding process.

Definition 2.12. For a given protocol, the set $\mathsf{valid}(\ell, q)$ of *valid* traces starting from (ℓ, q) is defined by induction on the length of the trace:

– $\varepsilon \in \mathsf{valid}(\ell, q)$
– $T \cdot m_i \in \mathsf{valid}(\ell, q)$ if $T \in \mathsf{valid}(\ell, q)$ and $d_i(\ell'_i) = m$, where $(\ell', q') = \Delta((\ell, q), T)$

An easy induction gives the following equivalent characterization of valid traces:

Proposition 2.13. *A trace T is in $\mathsf{valid}(\ell, q)$ iff for every decomposition $T = T' \cdot m_i \cdot T''$, we have $d_i(\ell'_i) = m$, where $(\ell', q') = \Delta((\ell, q), T')$ is the global state after executing T'.*

Starting from a global state (ℓ, q), we say that a valid trace is terminating if all the processes are decided after executing that trace. Formally, T is *terminating* if $\Delta((\ell, q), T) = (\ell', q')$ where $d_i(\ell'_i) \in \mathcal{O}$ for all $i \in [n]$. Notice here that we do not have to worry about processes that do not participate in the execution, since they will start with the input value $\perp$ and immediately "decide" the output $\perp \in \mathcal{O}$.

The following proposition states that computation cannot get stuck:

Proposition 2.14. *A valid trace is either terminating, or it can be extended to a longer valid trace.*

Proof. Assume T is not terminating, then in the global state $(\ell', q') = \Delta((\ell, q), T)$ there is some process i such that $d_i(\ell'_i) \in \mathcal{M}$. Taking $m = d_i(\ell'_i)$, we can easily check that the trace $T \cdot m_i$ is valid. □

We are almost ready to define what it means for a protocol to solve a task. Given a task $\Theta \subseteq \mathcal{I}^n \times \mathcal{O}^n$, and an assignment of input values $\bar{v} = (v_0, \dots, v_{n-1}) \in \mathsf{dom}\ \Theta$, we write $\iota(\bar{v}) \in \prod_i L_i$ for the tuple of local states obtained by applying the injection ι_i in each component: $\iota(\bar{v}) := (\iota_0(v_0), \dots, \iota_{n-1}(v_{n-1}))$. A global state is *initial* if it is of the form (ℓ, q_{init}), where $\ell = \iota(\bar{v})$ for some $\bar{v} \in \mathsf{dom}\ \Theta$.

We also need to define wait-free protocols. Just for the purpose of that definition, we extend the notion of valid traces to infinite traces: an infinite trace $T \in \mathcal{M}^\omega$ is *valid* starting from (ℓ, q) if all its finite prefixes are valid. Then, a protocol is *wait-free* if there is no valid infinite trace starting from an initial global state (ℓ, q_{init}). In a wait-free protocol, every valid computation path eventually leads to a terminating trace.

Definition 2.15. A wait-free protocol *solves* a task Θ if for every input $\bar{v} \in \mathsf{dom}\ \Theta$ (we write $\ell = \iota(\bar{v})$), and for every trace $T \in \mathcal{T}$ that is valid and terminating starting from (ℓ, q_{init}), we have $(\bar{v}, d(\ell')) \in \Theta$, where ℓ' is the local memory states after executing T, and $d(\ell')$ is the componentwise application of the decision functions d_i, i.e., $(\ell', q') = \Delta((\ell, q_{\text{init}}), T)$ and $d(\ell') = (d_0(\ell'_0), \dots, d_{n-1}(\ell'_{n-1}))$.

2.1.4 Protocol Complex

Now that we have defined a concrete notion of *solving* a task, our goal is to show that it agrees with the more abstract topological definition based on the existence of a simplicial map (Definition 1.29). To do that, the main technical difficulty is to carefully define the protocol complex that represents a given protocol.

Usually, the protocol complex is defined in terms of the notion of *view*. Informally, the view of a process at the end of an execution is the full history of all the events that occurred; it contains all the information that the process was able to observe during the computation. In [55], several definitions of view are thoroughly studied, in the context of the write-snapshot object. They show that the view can equivalently be characterized using execution traces; using interval orders; and using directed spaces. For other objects (e.g., test-and-set objects in [66], set-agreement objects in [65]), ad-hoc definition are usually given. For instance, for a protocol using only test-and-set flags, taking the sequence of all the observed values (0 or 1) of the flags that were tested during the execution seems to be a working notion of view.

Here, we will use the following very simple definition of view: the *view* of process i in some global state (ℓ, q) is its local state, ℓ_i. To build the protocol complex, we will be interested in particular in the views at the end of an execution, that is, the local states ℓ'_i, where (ℓ, q_{init}) is an initial global state, T is a terminating valid trace starting from (ℓ, q), and $(\ell', q') = \Delta((\ell, q_{\text{init}}), T)$.

Definition 2.16. The *protocol complex* associated to a protocol $\langle L_i, d_i, \tau_i \rangle_{i \in [n]}$ is the following chromatic simplicial complex:

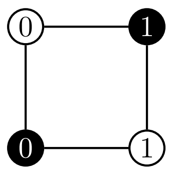
- Its i-colored vertices are all the possible views of process i after executing a valid and terminating trace T starting from any initial global state (ℓ, q_{init}).
- For every such valid and terminating trace T, there is a simplex whose vertices are the views of all the participating processes after executing T.

Remark 2.17. There is another candidate for the definition of view, following the intuition of "history of the computation". Given a trace T, each action m_i that occurs in T can be given a *response* value, as follows. We can decompose $T = T' \cdot m_i \cdot T''$, and write $q = \delta(q_{\text{init}}, T')$ for the internal state of the object after executing T'. The response associated to this occurrence of m_i in T is $\mathrm{ret}(q, m_i) \in \mathcal{V}$. Then the *history* of process i after executing the trace T is the (ordered) sequence of responses to all the actions m_i that occur in T. From this history, we can reconstruct the whole computation from the point of view of i.

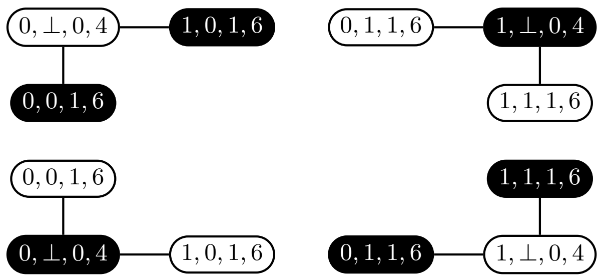
In general, history and view are not equivalent. Indeed, imagine a program where a process calls a shared object, but discards the value that is returned (formally, the transition function τ_i goes to the same local state for all values). Then, two different execution traces might produce different histories, but still have the same view (local state). However, these two notions become equivalent when we consider *full-information* protocols, that is, protocols where the graph of the transition function is a tree. Intuitively, that means that the processes are not allowed to forget information that they observed. In a full-information protocol, two traces with different histories will also have different views. Conversely, traces with the same histories will also have the same view (this works for any protocol).

To study task (un)solvability, we can safely restrict to full-information protocols [64]: indeed, if a task is solvable, then it is also solvable by a full-information protocol. In this context, the history would be a suitable notion of view. But in fact, there is no need for such a restriction. In the following, we work with general protocols, so the right notion of view is the one based on local states.

Example 2.18. We can draw the protocol complex for the protocol of Example 2.11. To avoid having too many vertices, we assume that there are only two possible input values, $v = 0$ or $v = 1$ (that is, we are solving *binary consensus*). The final views for process i are either of the form $(v, \bot, 0, 4) \in L_i$ (if the process wins the test-and-set race and ends at line 4), or of the form $(v, v', 1, 6) \in L_i$ (if the process loses the race and ends at line 6). In both cases, v is the input value of the process, and v' is the value read from the other process. In the picture below, the vertices of process P_0 are represented in black, and those of process P_1 are in white; the initial value / final view is written inside the nodes.



Input complex



Protocol complex

Before we can formulate our final result, we need to say a few words about the input and output complexes. As we said in Section 2.1.2, our notion of task is a direct reformulation of the usual simplicial definition; but for clarity, let us spell out the correspondence between the two. The vertices of the input complex are of the form (v, i), where $v \in \mathcal{I}$ is an input value with $v \neq \bot$, and $i \in [n]$ is a process number. Such a vertex is colored by i. For each input vector $\bar{v} \in \text{dom}\ \Theta$, we will have one input simplex, which is the set vertices (v_i, i) with $v_i \neq \bot$. This is indeed a simplicial complex thanks to the second condition of Definition 2.6. For the output complex, we do the same construction using the output values $v \in \mathcal{O}$ as vertices, and for simplices the codomain of Θ, $\text{codom}\ \Theta = \{\bar{v}' \in \mathcal{O}^n \mid \exists \bar{v}, (\bar{v}, \bar{v}') \in \Theta\}$. The task $\Theta \subseteq \mathcal{I}^n \times \mathcal{O}^n$ itself describes a rigid and chromatic carrier map: to an input vector $\bar{v} \in \text{dom}\ \Theta$, we associate a set of output vectors $\Theta(\bar{v})$, with the same set of participating processes by Definition 2.6. So, to the input simplex corresponding to $\bar{v}$, the task Θ associates a subcomplex of the output complex, which is pure and has the same dimension and colors.

Finally, remark that the vertices of our protocol complex are local states $\ell_i \in L_i$ such that the decision function d_i gives an output value $d_i(\ell_i) \in \mathcal{O}$. Hence, the functions $(d_i)_{i\in[n]}$ send the vertices of the protocol complex to the vertices of the output complex. Our final Theorem says that a protocol solves a task (in the sense of Definition 2.15) if and only if this map is a simplicial map, and it is carried by the task specification (cf. Definition 1.29).

Theorem 2.19. *The protocol $\langle L_i, d_i, \tau_i\rangle_{i\in[n]}$ solves the task Θ iff the decision functions $(d_i)_{i\in[n]}$ induce a chromatic simplicial map from the protocol complex to the output complex, which is carried by Θ.*

Proof (Sketch). Once we unfold the definitions, the statement of the theorem becomes almost tautological.

Assume the protocol solves the task and let X be a simplex in the protocol complex. We want to prove that $d(X)$ is a simplex of the output complex. By definition, there is an input vector $\bar{v} \in \text{dom}\ \Theta$, inducing an initial global state (ℓ, q_{init}) with $\ell = \iota(\bar{v})$, and a valid and terminating trace T starting from (ℓ, q_{init}), such that $\Delta((\ell, q_{\text{init}}), T) = (\ell', q')$ and X is the set of local states in ℓ' of the participating processes. The fact that the protocol solves the task says that $(\bar{v}, d(\ell')) \in \Theta$. So $d(\ell')$ belongs to the codomain of Θ, and thus $d(X)$ is a simplex of the output complex. This also implies that the simplicial map is carried by Θ.

Conversely, assume that the decision functions induce a simplicial map which is carried by Θ. So, for every $\bar{v} \in \operatorname{dom} \Theta$, and every valid and terminating trace T starting from $(\iota(\bar{v}), q_{\text{init}})$, we reach a final global state (ℓ', q') such that the simplex corresponding to ℓ' is sent to an output simplex in the codomain of Θ. For the processes that do not participate in the execution, the decision function will decide $\perp$ (by Definition 2.8), so we have $(\bar{v}, d(\ell')) \in \Theta$. This proves that the protocol solves the task. □

2.1.5 Limits of this approach

Theorem 2.19 extends the Asynchronous Computability Theorem of Herlihy and Shavit [70], which works only for read/write registers, by generalizing it to various other kinds of objects such as test-and-set (note that, in fact, it only generalizes one part of the asynchronous computability theorem – see the first paragraph of Section 2.4.4). However, as we already mentioned at the beginning of the section, Theorem 2.19 is still not as general as we would like. The reason is that our notion of object (Definition 2.1) is not expressive enough: it can only specify objects where each method of the object happens atomically. Many objects behave like that in practice; namely, all the *linearizable* implementations of concurrent data structures.

Nevertheless, many other objects that are relevant in distributed computability are not linearizable. For instance, Neiger has shown that set-agreement objects cannot be specified using such methods [99]; and yet, there is a paper that investigates task solvability with set-agreement objects, using the protocol complex approach [65]. Other examples of non-linearizable objects that are found in the field of distributed computability are investigated in [18]. In order to justify the simplicial complex approach in those cases, we need an even more general version of the Asynchronous Computability Theorem, which also applies to non-linearizable objects.

Remark 2.20. As mentioned earlier, this Section 2.1 is inspired from a paper by Goubault et al. [55]. In their paper, they study the immediate-snapshot object, which is not linearizable. This seems to contradict the above discussion, since they are able to specify it using a definition similar to Definition 2.1.

In fact, what they do is first specify the write-snapshot object as in Example 2.2. Then, they impose some restrictions on the shape of the execution traces that they consider: each process must alternate between writes and snapshots, and moreover there must be some "immediacy" condition. While this correctly models the immediate-snapshot computations, this is not a good way of specifying non-linearizable objects in general. Firstly, restricting the shape of execution traces breaks the assumption that the processes are asynchronous, i.e., that every interleaving of operations can occur. For immediate snapshot, it is known that this behavior can actually be implemented in an asynchronous way, but this would not be a safe assumption when considering general objects. Secondly, an immediate-snapshot operation essentially consists of two atomic operations: a write, followed by a snapshot. But if we want to study more complex objects, which might be performing thousands of atomic operations bundled together in one method call, we cannot keep splitting things up to the level of atomic operations: this would be like trying to analyze assembly code! Instead, we need a high-level notion of specification, which is able to express directly what an object is doing.

2.2 Specifying concurrent objects

What we did in Section 2.1 comes close to fulfilling the first goal of this chapter: we gave a concrete definition of what it means for a protocol to solve a task, and showed that we can recover from it the topological characterization of task solvability. This is already an extension of Herlihy and Shavit's Asynchronous computability Theorem [70], since it applies to objects such as test-and-set.

However, this is not completely satisfactory for a number of reasons.

(1) As we mentioned in Section 2.1.5, we are limited to objects whose operations seem to occur atomically. This includes many hardware primitives such as read/write registers, test-and-set, compare-and-swap, as well as linearizable implementations of common data structures such as lists or trees. We can even model non-atomic objects such as the immediate snapshot, by decomposing them into several atomic operations, and restricting to execution traces where these atomic operations appear in the right order. But this is not a good way of specifying the behavior of such objects: while the immediate snapshot object can be naturally decomposed in two atomic operations, it does not seem tractable to use the same procedure for more complex objects consisting of many atomic operations. Even worse, some objects such as the validity object or set-agreement objects do not seem to have a natural description in terms of atomic operations. What we would like is a more abstract, high-level specification, which is able to express what a complex object is doing without decomposing it into thousands of atomic operations.

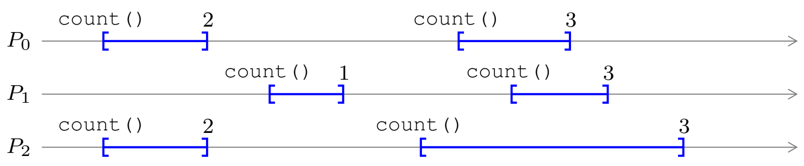
(2) Our notion of *object* (Definition 2.1) relies on an internal state of the object. While this is commonly seen in practice (for example, when specifying a list, the `push` and `pop` operations are usually described in terms of their effect on the internal state of the list), we would like to avoid this kind of confusion between implementation and specification. Indeed, in a distributed system, it is possible to implement a data structure without storing explicitly its internal state. This usually happens when we emulate a shared-memory architecture in a message-passing system [5, 76, 26]: the shared memory itself does not physically exist, but is disseminated among several processes. Thus, a more abstract way of specifying objects should not refer to a particular implementation of the object, but rather view it as a black box, hiding the internal state, and only specify how it interacts with its environment.

(3) Following the usual outline of the topological approach to distributed computing, we defined the two different notions of *tasks* and *objects*. This distinction between tasks and objects is what prevents us from formulating compositionality arguments such as: "if I can solve consensus using test-and-set, and I can solve set-agreement using consensus objects, then I can solve set-agreement using test-and-set". Given our current definitions, the consensus task and consensus objects are described using different formalisms, and the relationship between the two is unclear. However, the above reasoning seems intuitively sound: if I can implement a black box that solves consensus, and I can solve set-agreement using a black box that solves consensus, then I can solve set-agreement. A good notion of concurrent specification should be able to play both the role of the task ("what we want to implement") and the objects ("what we are allowed to use to compute").

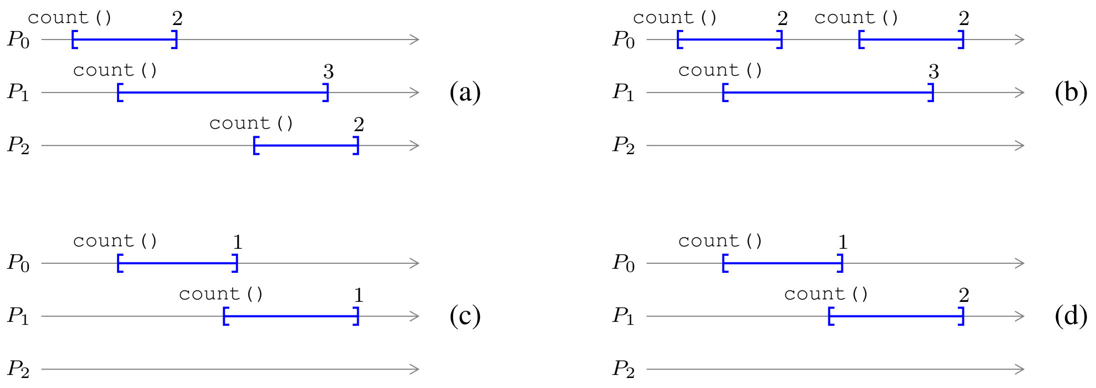
In short, the goal is now to repeat what we did in Section 2.1, but starting from a notion of concurrent specification which: (1) can specify objects with an intrinsically concurrent behavior; (2) does not refer to an internal state of the object; and (3) can specify both tasks and objects.

To illustrate what we mean by an object with an "intrinsically concurrent" behavior, let us first introduce a toy (and running) example that we call the COUNT object. Other examples include the immediate-snapshot object (Example 1.26), Java's exchanger object, as well as object versions of most tasks described in Section 1.3.1: k-set-agreement objects, the validity object, and the adopt-commit object. In short, any object that is *non-linearizable*. See [18] for more examples of non-linearizable objects that are useful in distributed computing.

Counting processes. When a process calls the COUNT object, it should return the number of processes that are currently calling the object in parallel. This is similar, although not completely identical, to Java's `Thread.activeCount` method, which returns "an estimate of" the current number of running threads. A typical execution of the COUNT object is depicted below. Each of the three processes P_0, P_1 and P_2 is calling the COUNT object twice. The horizontal axis represents the real-time scheduling of the operations and the intervals between square brackets depict the time span during which a process is executing the `count()` method: when two intervals vertically overlap, the processes are running the method concurrently. For instance, the last three calls are concurrent: the three processes should see each other and all return 3.



The specification of COUNT seems to be clear on this example. But what about the behaviors depicted below? In execution (a), process P_1 responds 3 since it has seen the two other processes; but there is no point in time when the three processes are running concurrently. Execution (b) represents the same situation, but this time P_1 has seen two different calls of P_0. In execution (c), the two calls are concurrent, but for a very short time, so they did not manage to see each other. In execution (d), process P_1 managed to see process P_0, but not reciprocally. All of these executions may or may not seem correct depending on what exact specification we have in mind. One could for example ask for a variant of the COUNT object that accepts executions (a) and (c), but rejects (b) and (d), and so on.



Thus, our description of the COUNT object has to be made more formal. We need a way to formally specify the behavior of a concurrent object without any ambiguity. This is what we are aiming to do in this section. More precisely, our definition of a *concurrent specification* will be given in Section 2.2.2, and will be crucial for the rest of the chapter. Then in Section 2.2.3, we compare our notion with other popular specification techniques, namely, linearizability and its variants.

Once we have a satisfying notion of concurrent specification, the next step will be to build a computational model out of it: what does it mean for a program to implement such a specification? And given a concurrent object that satisfies such a specification, what does it mean to use this object in a computation? Answering these questions will be the aim of Section 2.3.

2.2.1 Objects vs Tasks

In the classic setting of distributed computability, a fixed number of processes are communicating through shared *objects* in order to solve a *task*. As we mentioned in the introduction of this section, this point of view becomes problematic when we want to compose protocols. Moreover, there seems to be some overlap between the two notions: we sometimes want to think about the same entity either as a task or as an object, depending on the context. An instance of this, that we mentioned in Section 1.4, is the proof of Lo and Hadzilacos [88] that equality negation for two processes is not solvable using read/write registers. They argue that, assuming that the equality negation *task* is solvable using registers, then consensus could be solved using an equality negation *object*, and therefore consensus would be solvable using registers.

However, the difference between tasks and objects is not just a question of point of view. There are two main conceptual differences:

– Tasks are *one-shot*, whereas objects are *long-lived*. For tasks, all processes start together with their input values, compute, and once a process has decided an output value it does not take part in the computation anymore. This is contrasted with concurrent objects, where new processes can start and terminate at any point during the computation, and a single process is allowed to make several consecutive calls to the object.

– Tasks are "intrinsically concurrent", whereas objects have a "sequential flavor". Usually, a task is specified as a global relationship between the inputs and outputs of all the processes. On the other hand, an object should behave according to some sequential specification that it is supposed to emulate in a concurrent setting.

So, in the usual setting, tasks and objects are not comparable: there are objects that cannot be viewed as tasks, and vice versa.

Unifying objects and tasks. The usual way to specify an object (such as, for example, a concurrent list), is to rely on its standard sequential specification, and to extend it to a concurrent setting using a *correctness criterion*. There are many such criteria, such as sequential consistency [83], causal consistency [109] and linearizability [71]. These techniques are particularly suitable to specify objects such as lists or queues, which are concurrent versions of sequential data structures.

A first step towards unifying objects and tasks was the introduction of set-linearizability [99]: noticing that linearizability is unable to specify k-set-agreement objects, they defined a more expressive version of linearizability that allows intrinsically concurrent behaviors. However, this is still not sufficient to specify all tasks: in a recent paper [18], Castañeda et al. remarked that many interesting objects, such as *validity*, *write-snapshot* or *adopt-commit* are not set-linearizable. So, they introduced the notion of interval-linearizability, and proved that any task can be seen as the one-shot restriction of an interval-linearizable object. Therefore, the notion of interval-linearizability unifies tasks and objects: now, tasks can be seen as a subset of interval-linearizable objects. It is a strict subset, since tasks are one-shot, whereas objects can be used any number of times. More surprisingly, they also show that tasks are a strict subset of one-shot objects: the task formalism is not expressive enough to specify some one-shot behaviors of objects. They give an extended notion of tasks, called *refined tasks*, which corresponds exactly to the one-shot interval-linearizable objects.

In this thesis, we adopt a similar approach: our notion of object will be very general, in order to include all tasks. We do not use directly interval-linearizability to specify our objects, but as it turns out, our concurrent specifications will coincide with the interval-linearizable ones.

2.2.2 Concurrent specifications

In this section, we define the notion of concurrent object specification that we will use for the rest of the chapter. It is based on an idea introduced by Lamport in 1986 [85], which is expressive enough to specify non-linearizable objects. This fact was already remarked in Herlihy and Wing's original paper on linearizability [71], where they say the following about Lamport's notion of specification:

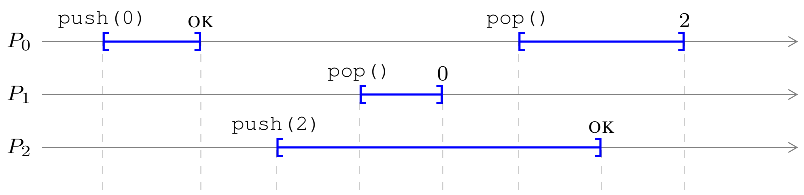
> *His approach is more general than ours, as it addresses [...] nonlinearizable as well as linearizable behavior. Our approach, however, focuses exclusively on a subset of concurrent computations that we believe to be the most interesting and useful. In place of a specification language powerful enough to specify all conceivable concurrent behaviors, we re-interpret assertions about "well-behaved" concurrent computations as assertions about their equivalent sequential computations.*

Here, our intention is precisely to have a notion of object which encompasses "all conceivable concurrent behaviors". So, Lamport's approach seems better suited than the ones based on linearizability.

Lamport's specifications. A simple but very general way of specifying concurrent objects was proposed by Lamport [85]. The specification of a concurrent object is simply the set of all the execution traces that we consider correct for this object. For the COUNT object example that we described in the beginning of the section, if we draw every possible diagram such as the ones in the pictures (a) – (d) and decide which ones correspond to "good" behaviors and which ones do not, we will have a well-specified object. Notice that this point of view allows us to specify a concurrent object without referring to a notion of internal state. For example, to specify a concurrent list using this method, we never mention a chain of memory cells pointing to each other: all we need to say is how the `push` and `pop` operations will interact when they are called successively.

Of course, we need a mathematical abstraction of the drawings (a) – (d) representing executions of the program. We represent them by *execution traces*, i.e., finite words whose letters are either *invocations* i_i^x or *responses* r_i^y. In the picture (e) below, an execution of a concurrent list is depicted. The corresponding trace T has events of the form $\mathsf{i}_i^{\mathtt{push},x}$ representing the invocation of the `push` method by process i with input value x, and r_i^y representing the response of some operation of process i with output value y.

Remark 2.21. Lamport originally axiomatized the happens-before partial order between operations in order to formally define executions; the trace formalism that we use here was introduced by Herlihy and Wing in their linearizability paper [71]. These two formalisms are not equivalent: for example, if we permute two invocation events, we obtain different traces but the happens-before relation is unchanged. This distinction will vanish when we impose some axioms on our specifications in Definition 2.22: as we will see, two successive invocations (or two successive responses) will always commute.



(e)

$$T \quad = \quad \mathsf{i}_0^{\mathrm{push},0} \cdot \mathsf{r}_0^{\mathrm{OK}} \cdot \mathsf{i}_2^{\mathrm{push},2} \cdot \mathsf{i}_1^{\mathrm{pop}} \cdot \mathsf{r}_1^{0} \cdot \mathsf{i}_0^{\mathrm{pop}} \cdot \mathsf{r}_2^{\mathrm{OK}} \cdot \mathsf{r}_0^{2}$$

An implicit assumption of this representation is that invocations and responses are totally ordered according to some global time clock. At first sight this might look like a harmless assumption, but recent works have been trying to get rid of this hypothesis, with applications to weak memory models [115] and relativistic effects [45]. Another assumption is that only the relative position of the events in time matters: the intervals can be moved around as long as the overlapping pattern is conserved. This means for example that we cannot express specifications based on timing constraints. Another consequence, for our COUNT object example, is that we cannot tell how long processes overlap, so execution (c) is indistinguishable from an execution where P_0 and P_1 are fully concurrent.

Concurrent specifications. The notion of concurrent specification that we will use in the rest of this chapter is very close to Lamport's idea: it will be a set of execution traces (intended to be the "correct" ones), but we also impose a number of properties that this set of traces has to satisfy. In particular, the crucial one is the so-called *expansion* property, which will be necessary to prove most of the important results of the chapter.

We now suppose fixed a number $n \in \mathbb{N}$ of *processes* and write $[n] = \{0, 1, \dots, n-1\}$ for the set of *process names* (a process is identified by its number). We also suppose fixed a set $\mathcal{V}$ of *values* that can be exchanged between processes and objects (typically $\mathcal{V} = \mathbb{N}$). The set $\mathcal{A}$ of possible *actions* for an object is defined as

$$\mathcal{A} \quad = \quad \{\mathsf{i}_i^x \mid i \in [n], x \in \mathcal{V}\} \cup \{\mathsf{r}_i^y \mid i \in [n], y \in \mathcal{V}\}$$

An action is thus either

- i_i^x: an *invocation* of the object by the process i with input value x,
- r_i^y: a *response* of the object to the process i with output value y.

An *execution trace* is a finite sequence of actions, i.e., an element of $\mathcal{A}^*$; we write ε for the empty trace and $T \cdot T'$ or TT' for the concatenation of two traces T and T'.

Given a process $i \in [n]$, the i-th *projection* $\pi_i(T)$ of a trace $T \in \mathcal{T}$ is the trace obtained from T by removing all the actions of the form i_j^x or r_j^x with $j \neq i$. A trace $T \in \mathcal{T}$ is *alternating* if for all $i \in [n]$, $\pi_i(T)$ is either empty or it begins with an invocation and alternates between invocations and responses, i.e., using the traditional notation of regular expressions:

$$\pi_i(T) \quad \in \quad \Big(\bigcup_{x,y \in \mathcal{V}} \mathsf{i}_i^x \cdot \mathsf{r}_i^y\Big)^* \cdot \Big(\bigcup_{x \in \mathcal{V}} \mathsf{i}_i^x + \varepsilon\Big)$$

We write $\mathcal{T} \subseteq \mathcal{A}^*$ for the set of alternating traces. In the remaining of the chapter, we will only consider alternating traces, and often drop the adjective "alternating". If $\pi_i(T)$ ends with an invocation, we call it a *pending* invocation. An alternating trace T is *complete* if it does not have any pending invocation.

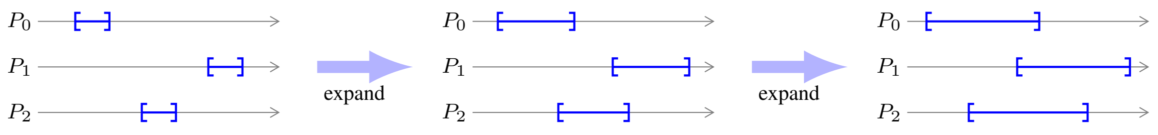
Definition 2.22. A *concurrent specification* σ is a subset of $\mathcal{T}$ which is

(1) *prefix-closed*: if $T \cdot T' \in \sigma$ then $T \in \sigma$,
(2) *non-empty*: $\varepsilon \in \sigma$,
(3) *receptive*: if $T \in \sigma$ and $\pi_i(T)$ has no pending invocation, then $T \cdot \mathsf{i}_i^x \in \sigma$ for every $x \in \mathcal{V}$,
(4) *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$,
(5) is closed under *expansion*:
   – if $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T' \in \sigma$ where a_j is an action of process $j \neq i$, then $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T' \in \sigma$.
   – if $T \cdot \mathsf{r}_i^y \cdot a_j \cdot T' \in \sigma$ where a_j is an action of process $j \neq i$, then $T \cdot a_j \cdot \mathsf{r}_i^y \cdot T' \in \sigma$.

We write CSpec for the set of concurrent specifications.

A concurrent specification σ is the set of all executions that we consider acceptable: a program *implements* the specification σ if all the execution traces that it generates belong to σ (this will be detailed in Section 2.3). The axioms (1-4) are quite natural and commonly considered in the literature (e.g. in [71]). They can be read as follows: (1) an object does one action at a time, i.e., the processes are asynchronous, (2) an object can do nothing, (3) it is always possible to invoke an object. The axiom (4) states that objects always answer and in a non-blocking way; this is a less fundamental axiom and more of a design choice, since we want to model wait-free computation. Note that receptivity (3) does not force objects to accept all inputs: we could have a distinguished "error" value which is returned in case of an invalid input. Similarly, an object with several inputs, or several interacting methods (for example, a list) can be modeled by choosing a suitable set of values $\mathcal{V}$.

The condition (5) might seem more surprising, and will be crucial in the rest of the chapter. As the name suggests, it can be understood pictorially as follows. Given some execution trace that is considered correct, if we *expand* the intervals then the new trace must also be correct. When we expand the intervals, more overlappings between them might occur: the trace becomes more concurrent.



It is easy to see that the expansion property implies that invocations "commute": we have $T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in \sigma$ *iff* $T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$, and similarly for responses. For example, one could expect to specify an object whose behavior depends on which process was invoked first; but that would break the commutativity of invocations. As we show in Section 2.3, in an asynchronous model, no program can implement such a specification. Let us explain intuitively why this "expansion property" should hold. Suppose $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T'$ is an acceptable execution of the object we are specifying. Then, in the trace $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T'$, where process i invokes its operation a little bit earlier, i might be idle for a while and only start computing after the action a_j has occurred, resulting in the same behavior. Thus, the execution $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T'$ should also be considered acceptable. The second condition is similar: in the execution trace $T \cdot a_j \cdot \mathsf{r}_i^y \cdot T'$, it might be the case that process i had already finished its computation before the action a_j occurred, but then it was idle for a while before returning its output. So, it might actually behave like in the trace $T \cdot \mathsf{r}_i^y \cdot a_j \cdot T'$, which we assumed to be correct. Thus, this expansion condition reflects the idea that invocations and responses do not correspond to actual actions taken by the processes, but rather define an interval in which they are allowed to take steps.

Example 2.23. Applied to the COUNT object example, the expansion property implies that execution (c) *must* be accepted, since it is obtained by expanding a correct sequential execution. This might seem surprising: our first intuition was that execution (c) is a mistake, where the processes did not manage to see each other. But as we argue in the next sections, the expansion property is an essential property that any concurrent specification must satisfy, just as prefix-closure. This means that the "mistake" of execution (c) cannot be avoided; we cannot force two concurrent processes to see each other, there is always a chance that they might exhibit sequential behavior.

As we said earlier, the condition (5) above ensures that two invocations (resp. two responses) "commute": we say that two alternating traces $T, T' \in \mathcal{T}$ are *equivalent*, written $T \equiv T'$, if one is obtained from the other by reordering the actions within each block of consecutive invocations or consecutive responses. Generally, in the rest of the chapter, we are only interested in studying traces up to equivalence.

It will prove quite useful to consider operations in traces, which are pairs consisting of an invocation and its matching response. Formally,

Definition 2.24. Consider an alternating trace $T = T_0 \cdots T_{k-1}$. An *operation* of process i in T is either

- a *complete operation*: a pair (p, q) such that $T_p = \mathsf{i}_i^x$ and $T_q = \mathsf{r}_i^y$ and T_q comes right after T_p in $\pi_i(T)$, or
- a *pending operation*: a pair $(p, +\infty)$ where T_p is a pending invocation,

where $p, q \in \mathbb{N}$, with $0 \leq p, q < k$, are indices of actions in the trace. We write $\mathrm{op}_i(T)$ for the set of operations of the i-th process and $\mathrm{op}(T)$ for the set of all operations.

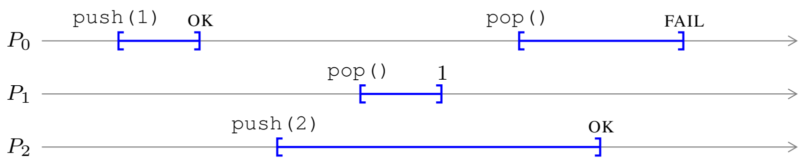
The operations of a trace can be ordered by the smallest partial order $\preceq$ such that $(p, q) \prec (p', q')$ whenever $q < p'$. This partial order is called *precedence* and two incomparable operations are called *overlapping* or *concurrent*. Note that for every $i \in [n]$, $(\mathrm{op}_i(T), \preceq)$ is totally ordered.

2.2.3 Comparison of linearizability-based techniques

The notion of concurrent specification of Definition 2.22 is very general, in the sense that it can express all the concurrent objects that we are interested in studying:

- concurrent implementations of sequential data structures, such as lists, trees, and so on;
- hardware communication primitives: read/write registers, test-and-set, message-passing;
- and most importantly, the objects that are studied in fault-tolerant distributed computing, such as immediate-snapshot, consensus or set-agreement objects, the COUNT object. These kinds of objects are usually "intrinsically concurrent", meaning that the concurrent traces might exhibit different behavior than the sequential ones.

However, Definition 2.22 is not very well suited for reasoning about concurrent objects in practice. For example, if we want to specify what is a concurrent list, we need to find a way to enumerate all the correct execution traces. Without additional tools, this seems quite tedious: even if we consider just one concurrent trace, such as the one below, it can already be quite tricky to figure out whether it is a correct execution or not.



Trying to describe formally the general form of all the correct execution traces seems like an awkward and unintuitive way to think about concurrent lists. Moreover, there is one fact about lists that we did not use. In the sequential setting, there are many well-studied techniques that can specify how a list should behave. For instance, we could use an automata-like representation of a list, like we did in Section 2.1.1; or rely on a more involved specification method such as Hoare's logic [73] or temporal logic [102]. Thus, the set of *sequential* executions of a concurrent list is easy to describe. And since the role of the concurrent executions is simply to emulate these sequential behaviors, we can expect to be able to define the concurrent traces in terms of the sequential ones.

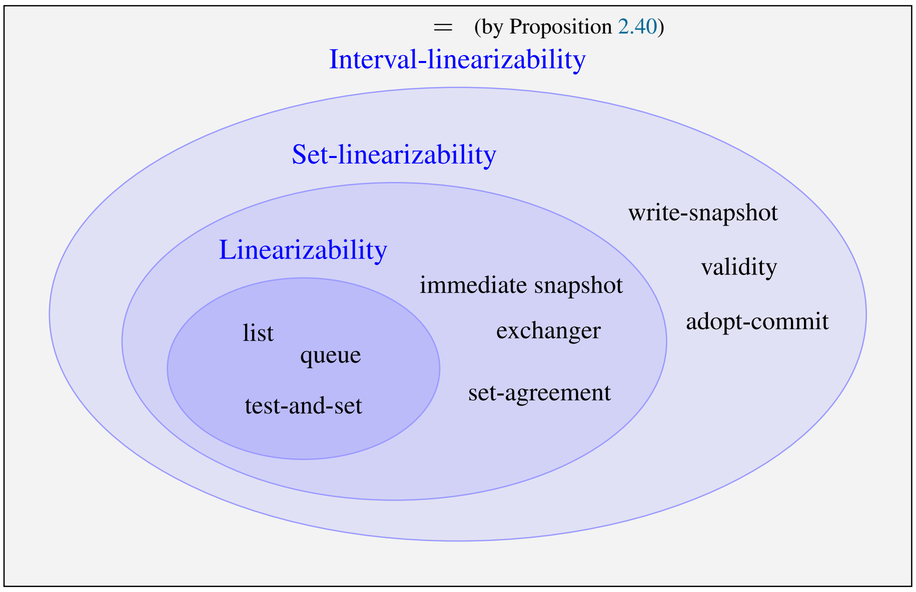
This specificity of concurrent lists actually applies to many other concurrent data structures: usually, we have a sequential object that we would like to be able to use in a concurrent setting, while keeping the illusion that the operations occurred sequentially. Many techniques have been developed in order to specify such objects: atomicity [94], sequential consistency [83], serializability [101], causal consistency [109], or linearizability [71]. In general, we are given a *sequential specification* σ, and the goal is to find a *correctness criterion* which, given a concurrent trace, says whether it is correct or not with respect to σ. Depending on the correctness criterion that we choose, we obtain different objects: for instance, a sequentially consistent concurrent list or a linearizable concurrent list are two distinct concurrent objects. If we see them as sets of correct traces, as in Definition 2.22, the former contains strictly more traces than the latter: it is a more relaxed version of concurrent lists, which allows more behaviors to occur.

In this section, we will focus on one of those correctness criteria, *linearizability*, which was introduced by Herlihy and Wing [71]. It is very popular thanks to its *locality* property, which allows programmers to reason modularly about a complex system by studying each of its components in isolation. Here, we adopt a slightly unusual point of view on linearizability: instead of seeing it as a correctness criterion, we view it as a map that turns a sequential specification into a concurrent specification. Namely, given a sequential specification σ, we are interested in studying the set $\mathsf{Lin}(\sigma)$ of all the *linearizable* traces (w.r.t. σ), which are to be understood as the correct ones. We will show that is it a concurrent specification in the sense of Definition 2.22. Thus, Lin is a map from the set of sequential specifications to the set of concurrent specifications. By studying the properties of this map, we obtain Theorem 2.32, which explains in what sense linearizability is the canonical way to extend a sequential specification to a concurrent one.

Linearizability is a very powerful tool when we want to extend sequential data structures to a concurrent setting. As we discussed previously, we also want to consider objects that exhibit a more concurrent behavior. The original paper on linearizability [71] already remarked that there are non-linearizable objects; that is, objects whose behavior cannot be specified using linearizability. The COUNT object described in Section 2.2 is a typical example of such an object. Many other examples can be found in the area of distributed computability [64], Another notable example is Java's *Exchanger* object. In order to specify those objects, many variants of linearizability have been defined: set-linearizability [99] (a.k.a. concurrency-aware linearizability [60]), local linearizability [57], and interval-linearizability [18]. Here, we will focus on two of them, which are relevant in the context of fault-tolerant distributed computing. Set-linearizability was introduced by Neiger in order to specify set-agreement objects. Later on, Castañeda et al. remarked that the (non-immediate) write-snapshot object and the validity object are not set-linearizable, and defined the notion of interval-linearizability to be able to specify them. The latter notion is the most expressive one; in particular, they show that their framework allows to specify every concurrent *task*.

As we will see in Proposition 2.27, our Definition 2.22 encompasses all those specification techniques based on linearizability. As it turns out (Proposition 2.40), we actually have an equality between CSpec and the set of interval-linearizable specifications.

Concurrent specifications CSpec



As in the case of standard linearizability, all these techniques give rise to a concurrent specification $\mathsf{Lin}(\sigma) \in \mathsf{CSpec}$ (in the sense of Definition 2.22); but here, σ is no longer a sequential specification. Namely, σ specifies the sequential behaviors of the object, but also *some* of the concurrent ones. The role of the Lin map is to extend this σ in order to specify *all* concurrent behaviors.

For ease of presentation, we begin by introducing a general notion called $\mathcal{L}$-linearizability, which subsumes the three variants of linearizability that we have discussed. It is actually quite straightforward to check that the usual notions of linearizability, set-linearizability and interval-linearizability are recovered by instantiating $\mathcal{L}$ with the right set of traces. Thus, the definition of $\mathcal{L}$-linearizability itself should not be regarded as a new concept; it is merely a presentation trick to avoid dealing with three variants of the same definition. The main result of this section is stated in Theorem 2.32. We then explore its consequences in three particular cases: standard, set- and interval-linearizability.

$\mathcal{L}$-specifications

We first introduce a notion of specification, akin to the one of Definition 2.22, which is parameterized by a set $\mathcal{L}$ of traces, which we abstractly consider as the "linear" ones. To recover the standard notion of linearizability, we let $\mathcal{L}$ be the set of all sequential traces, in which case $\mathcal{L}$-specifications correspond to sequential specifications (this will be done formally later).
We always require $\mathcal{L}$ to be prefix-closed, non-empty and

- *receptive* for complete traces: if $T \in \mathcal{L}$ is complete then $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ for all $i \in [n]$ and $x \in \mathcal{V}$,
- *fully total*: if $T \in \mathcal{L}$ and $\pi_i(T)$ has a pending invocation then $T \cdot \mathsf{r}_i^x \in \mathcal{L}$ for $\underline{\text{every}}$ $x \in \mathcal{V}$.

Intuitively, these conditions mean that $\mathcal{L}$ is a set of traces that have some specific "shape" (e.g., being sequential), but it does not say anything about the input or output values. The set of sequential traces is the smallest set satisfying those properties. Now, given such a set $\mathcal{L}$, an $\mathcal{L}$-specification says which execution traces are correct or not, but only among those of $\mathcal{L}$.

Definition 2.25 ($\mathcal{L}$-specification)**.** An *$\mathcal{L}$-specification* σ is a set of traces in $\mathcal{L}$ which is

(1) *prefix-closed*,
(2) *non-empty*,
(3') *receptive within* $\mathcal{L}$: for all $T \in \sigma$, $i \in [n]$ and $x \in \mathcal{V}$, if $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ then $T \cdot \mathsf{i}_i^x \in \sigma$,
(4) *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$.

We write $\mathsf{Spec}_{\mathcal{L}}$ for the set of $\mathcal{L}$-specifications.

Notice that conditions (1), (2), (4) are the same as in Definition 2.22, and condition (3') is just a weakened version of (3) which only applies to traces in $\mathcal{L}$. In particular, a concurrent specification (Definition 2.22) is an $\mathcal{L}$-specification for $\mathcal{L} = \mathcal{T}$ the set of all traces, such that moreover the expansion property (5) is satisfied.

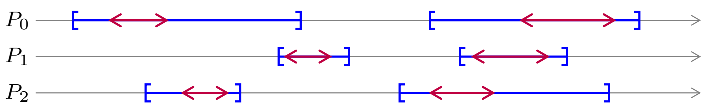
$\mathcal{L}$-linearizability

We now define a notion of linearizability which is parameterized by some set $\mathcal{L}$. So, assume we are given an $\mathcal{L}$-specification $\sigma \in \mathsf{Spec}_{\mathcal{L}}$, which only specifies the behavior of the object among the traces of $\mathcal{L}$. Our goal is to extend it to a full concurrent specification $\mathsf{Lin}_{\mathcal{L}}(\sigma)$. For this, we need a correctness criterion that says, for every concurrent trace $T \in \mathcal{T}$, whether it is a good or a bad behavior with respect to σ. This criterion is precisely the notion of $\mathcal{L}$-linearizability, that we now define.

We write $T \rightsquigarrow T'$ when the trace T' can be obtained from the trace T by applying the following series of local transformations (forming a string rewriting system):

$$\mathsf{i}_i^x \cdot \mathsf{i}_j^y \rightsquigarrow \mathsf{i}_j^y \cdot \mathsf{i}_i^x \qquad \mathsf{r}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{r}_i^x \qquad \mathsf{i}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{i}_i^x \qquad \text{for } i \neq j.$$

This amounts to contracting the intervals, the opposite of expansion. Two traces T and T' such that $T \rightsquigarrow T'$ should be pictured as follows, where T is represented in blue with square brackets, T' in red with angle brackets:



Definition 2.26. A trace $T \in \mathcal{T}$ is *$\mathcal{L}$-linearizable* w.r.t. an $\mathcal{L}$-specification σ if there exists a completion T' of T, obtained by appending responses to the pending invocations of T, and a trace $S \in \sigma$, such that $T' \rightsquigarrow S$. Moreover, we write $\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \mathcal{T}$ for the set of $\mathcal{L}$-linearizable traces.

The above definition in terms of a rewriting system is quite unusual, but it is not difficult to check that it is equivalent to the usual one of [71], which uses the precedence partial order $(\mathrm{op}(T), \preceq)$. This local presentation as a rewriting system is sometimes used in linearizability proofs using Lipton's left/right movers [87]. For the sake of completeness, we will prove in the next paragraph that these two alternative ways of defining linearizability are equivalent. But first, let us prove that $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ is indeed a concurrent specification; i.e., that it satisfies the axioms (1-5) of Definition 2.22.

Proposition 2.27. *Let $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ be an $\mathcal{L}$-specification. Then*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \quad = \quad \{T \in \mathcal{T} \mid T \text{ is } \mathcal{L}\text{-linearizable with respect to } \sigma\}$$

is a concurrent specification.

Proof. Let σ be an $\mathcal{L}$-specification. We check that $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ satisfies all the conditions of concurrent specifications (Definition 2.22).

(1) Prefix-closure is a bit more technical than other conditions, see below.

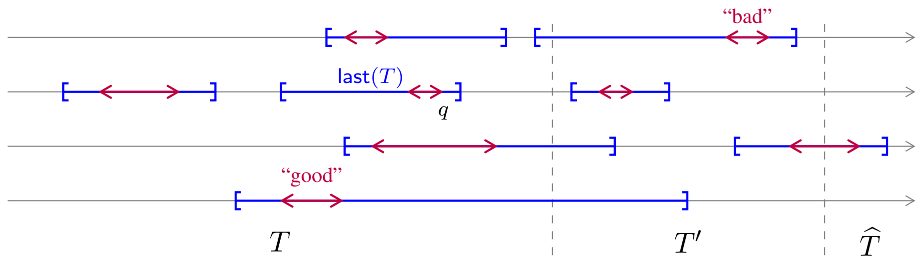
(2) The empty trace is linearizable.

(3) For receptivity, let $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$ be a linearizable trace such that $\pi_i(T)$ does not have a pending invocation. Let $T \cdot \widehat{T}$ be an extension of T and $S \in \sigma$ such that $T \cdot \widehat{T} \rightsquigarrow S$. We want to show that $T \cdot \mathsf{i}_i^x$ is linearizable. Since σ is an $\mathcal{L}$-specification and S is complete, by receptivity $S \cdot \mathsf{i}_i^x \in \sigma$, and by totality there exists $y \in \mathcal{V}$ such that $S' = S \cdot \mathsf{i}_i^x \cdot \mathsf{r}_i^y \in \sigma$. Take $T' = T \cdot \mathsf{i}_i^x \cdot \widehat{T} \cdot \mathsf{r}_i^y$ as the extension of $T \cdot \mathsf{i}_i^x$ required in Definition 2.26, then we can check that $T' \rightsquigarrow S'$.

(4) For totality, let $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$ be a linearizable trace such that $\pi_i(T)$ has a pending invocation. Again, we write $T \cdot \widehat{T}$ and $S \in \sigma$ the two traces of Definition 2.26. The suffix $\widehat{T}$ contains some response r_i^y, and by reordering the responses in $\widehat{T}$ we get $T \cdot \widehat{T} \equiv T \cdot \mathsf{r}_i^y \cdot \widehat{T}'$. So we obtain an extension of $T \cdot \mathsf{r}_i^y$ such that $T \cdot \mathsf{r}_i^y \cdot \widehat{T}' \rightsquigarrow S$.

(5) For closure under expansion, suppose $T = U \cdot a_j \cdot \mathsf{i}_i^x \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$ with $i \neq j$. So there is a sequence of responses $\widehat{T}$ and a trace $S \in \sigma$ such that $T \cdot \widehat{T} \rightsquigarrow S$. We want to show that $T' = U \cdot \mathsf{i}_i^x \cdot a_j \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. By rewriting one step, we have $T' \rightsquigarrow T$. So, we pick the same suffix $\widehat{T}$ and trace $S \in \sigma$, and since $T' \cdot \widehat{T} \rightsquigarrow T \cdot \widehat{T} \rightsquigarrow S$, we are done. The same reasoning applies when starting from $T = U \cdot \mathsf{r}_i^y \cdot a_j \cdot V$.

As mentioned above, closure under prefix is a bit more tedious to check. Suppose $T \cdot T' \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. There is an extension $T \cdot T' \cdot \widehat{T}$ that has a linearization $S \in \sigma$. We want to show that T is linearizable. First, we need to extend T by adding responses to its pending invocations. These invocations have responses either in T' or in $\widehat{T}$: a first idea would be to use these responses. But some of these responses might be "bad" in the sense that they rely on actions of T' in order to be valid. Some of them might be "good" in the sense that some actions of T rely on them in order to valid. The picture below shows the trace $T \cdot T' \cdot \widehat{T}$ (in blue with squared brackets) and a linearization S (in red with angle brackets):



Given an operation $e \in \mathrm{op}(T)$, we can view it as an operation of S, whose response happens at index $\phi(e)$ in S. Let $\mathsf{last}(T) \in \mathrm{op}(T)$ be the complete operation of T whose index $\phi(\mathsf{last}(T))$ is maximal, and let q be that index. We truncate S after index q to obtain $S' = S_0 \cdots S_q$. Then $S' \in \sigma$ since σ is prefix-closed. S' might have some pending invocations, but we can complete it using totality to obtain a trace $S'' \in \sigma$. Then all the complete operations of T are in S'', by definition of the index q. Moreover, all the other operations of S'' correspond to pending invocations of T (they are the "good" responses we want to keep). Indeed, an operation e whose invocation is not in T satisfies $\mathsf{last}(T) \prec_{TT'\widehat{T}} e$, which implies $\mathsf{last}(T) \prec_S e$, and so $e \notin \mathrm{op}(S'')$. Write $\bar{T}$ for the trace T where the pending invocations whose operations are not in $\mathrm{op}(S'')$ (the "bad" ones) have been removed. Complete $\bar{T}$ by appending responses to match those of S'', and we can check that S'' is a linearization of $\bar{T}$. So, $\bar{T} \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. Since we have already proved receptivity, we can add the missing invocations at the end of $\bar{T}$, and then we move them inside to their original place using expansions and commutations, we finally get $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. □

Therefore, by Proposition 2.27, we have a map $\mathsf{Lin}_{\mathcal{L}} : \mathsf{Spec}_{\mathcal{L}} \to \mathsf{CSpec}$. This result shows that all the axioms that we impose on our concurrent specifications are reasonable. Indeed, when we instantiate $\mathcal{L}$ in order to recover the standard notion of linearizability (as well as set- and interval-linearizability), this will imply that every object that can be specified using linearizability, is in CSpec. In short, all our axioms are naturally enforced by all the specification techniques based on linearizability.

The usual definition of linearizability

We have defined the notion of $\mathcal{L}$-linearizability in a slightly unusual way, using a rewriting system. We detail here how to relate it to the usual definition given in [71]. To make it fully formal, we need some preliminary definitions.

Definition 2.28. Two alternating traces $T, T' \in \mathcal{T}$ are *compatible* if for all $i \in [n]$, $\pi_i(T) = \pi_i(T')$.

When two traces T and T' are compatible, T' is a reordering of the actions of T that does not exchange actions with the same process number. Thus, there is a canonical bijection between $\mathrm{op}(T)$ and $\mathrm{op}(T')$ that sends the k-th operation of process i in T to the k-th operation of process i in T'. In the following, we keep this bijection implicit and we work as if $\mathrm{op}(T) = \mathrm{op}(T')$.

Definition 2.29. A trace T' is a *linearization* of T if:

- T and T' are compatible, and
- if $e \preceq_T e'$ in $\mathrm{op}(T)$, then $e \preceq_{T'} e'$ in $\mathrm{op}(T')$.

The original definition of linearizability is just like Definition 2.26, except that instead of $T' \rightsquigarrow S$, we require S to be a linearization of T'. Thus, we need to show that these two notions coincide.

Lemma 2.30. *Two traces are equivalent iff they are compatible and* $\preceq_T = \preceq_{T'}$.

Proof. Remember that $T \equiv T'$ when they only differ by a reordering of consecutive invocations (resp. consecutive responses). Two equivalent traces are compatible because of alternation: they cannot contain two consecutive invocations (or responses) with the same process number. Moreover, the precedence ordering is preserved since it only compares the position of an invocation with the position of a response. Conversely, assume T and T' are compatible but not equivalent. That means an invocation and a response have been exchanged: let e and e' be the two corresponding operations ($e = e'$ would break compatibility). Suppose w.l.o.g. that the invocation of e happens after the response of e' in T and are reversed in T', then $e' \preceq_T e$ but $e' \not\preceq_{T'} e$. □

If $T = U \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot V$ and $T' = U \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot V$ for $i \neq j$, we say that T' is obtained from T by *one-step contraction*, and we write $T \overset{1}{\rightsquigarrow} T'$.

Lemma 2.31. *T' is a linearization of T iff $T \rightsquigarrow T'$, i.e. we can go from T to T' by a sequence of contractions and equivalences:*

$$T \equiv T_0 \overset{1}{\rightsquigarrow} T_1 \equiv T_1' \overset{1}{\rightsquigarrow} \cdots \overset{1}{\rightsquigarrow} T_k \equiv T'$$

Proof. The right-to-left implication is due to the fact that $T \overset{1}{\rightsquigarrow} T'$ implies $\preceq_T \subseteq \preceq_{T'}$. Indeed, two overlapping intervals have been separated, resulting in one more couple in the precedence relation.

For the converse, if $\preceq_T = \preceq_{T'}$, we are done by Lemma 2.30. So assume that we have a strict inclusion $\preceq_T \subsetneq \preceq_{T'}$. We will construct a trace T_1 such that $\preceq_T \subsetneq \preceq_{T_1} \subseteq \preceq_{T'}$ and $T \overset{1}{\rightsquigarrow} T_1$ (up to equivalence). Since $\preceq_T \subsetneq \preceq_{T'}$, there are two operations e, e' such that $e \preceq_{T'} e'$ but e and e' are incomparable in $\mathrm{op}(T)$. Among such pairs (e, e'), we choose one such that the difference $q - p'$ is minimal, where $e = (p, q)$ and $e' = (p', q')$, with indexes in T.

Claim: Up to equivalence, T is of the form $U \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot V$ where r_i^x is the response of e and i_j^y is the invocation of e'. Indeed, assume by contradiction that T is of the form $U \cdot \mathsf{i}_i^x \cdots \mathsf{r}' \cdots \mathsf{i}' \cdots \mathsf{r}_j^y \cdot V$, then we have two candidates for a smaller gap between invocation and response; a case analysis shows that at least one of them would work, thus contradicting minimality.

By applying one step of contraction, we obtain $T_1 = U \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot V$. The ordering $\preceq_{T_1}$ is $\preceq_T$ with one more relation $e \preceq e'$.

Starting from T, we repeat this reasoning until we reach T'. This procedure terminates since the sequence of orderings $(\preceq_{T_k})_k$ is strictly increasing and there are only finitely many such relations (because $\mathrm{op}(T)$ is finite). □

Lemma 2.31 almost bridges the gap between our definition of linearizability and the original one. There remains one minor difference: the completion T' of T of Definition 2.26 is usually allowed to remove some of the pending invocations of T. This is not useful here since all our specifications are total and non-blocking: a response to the pending invocations can always be found. In the rest of the chapter, we always use the definition based on rewriting. It is quite convenient to be able to work directly on the trace instead of its associated partial order; moreover, the $\rightsquigarrow$ relation is a straight counterpart of the expansion property (5) of Definition 2.22.

Comparison with concurrent specifications

As we saw in Proposition 2.27, we have defined a map $\mathsf{Lin}_{\mathcal{L}} : \mathsf{Spec}_{\mathcal{L}} \to \mathsf{CSpec}$. In order to compare $\mathcal{L}$-specifications and concurrent ones, we define a map in the other direction, $\mathsf{U}_{\mathcal{L}} : \mathsf{CSpec} \to \mathsf{Spec}_{\mathcal{L}}$. Its definition is straightforward. Given a concurrent specification τ, we keep only the linear traces of τ, i.e., $\mathsf{U}_{\mathcal{L}}(\tau) := \tau \cap \mathcal{L}$. Intuitively, the map $\mathsf{U}_{\mathcal{L}}$ forgets about what the object τ does on the traces that are not linear. The role of $\mathsf{Lin}_{\mathcal{L}}$ is then to try to reconstruct these missing traces; but of course, in general, some information might have been lost, and $\mathsf{Lin}_{\mathcal{L}}(\mathsf{U}_{\mathcal{L}}(\tau)) \neq \tau$.

The precise relationship between $\mathsf{Lin}_{\mathcal{L}}$ and $\mathsf{U}_{\mathcal{L}}$ is contained in Theorem 2.32 below: they form a Galois connection. We can picture it as in the following diagram:

$$\begin{tikzcd} \mathsf{Spec}_{\mathcal{L}} \arrow[r, bend left, "\mathsf{Lin}_{\mathcal{L}}"] \arrow[r, phantom, "\perp"] & \mathsf{CSpec} \arrow[l, bend left, "\mathsf{U}_{\mathcal{L}}"] \end{tikzcd}$$

This important property explains in which sense linearizability is a canonical way of turning a weaker specification σ (e.g., one specifying only sequential behaviors) into a concurrent specification. It is the main result of this section.

Theorem 2.32. *The functions* $\mathsf{Lin}_{\mathcal{L}}$ *and* $\mathsf{U}_{\mathcal{L}}$ *are monotonous w.r.t. inclusions, and form a Galois connection: for every* $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ *and* $\tau \in \mathsf{CSpec}$*,*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau \qquad \Longleftrightarrow \qquad \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau).$$

Proof. The monotonicity of $\mathsf{U}_{\mathcal{L}}$ is trivial. For $\mathsf{Lin}_{\mathcal{L}}$, assume $\sigma \subseteq \sigma'$ and $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. Let $S \in \sigma$ be a linearization of T. Since $S \in \sigma'$, we also have $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma')$.

Now let σ and τ as in the theorem and assume $\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau$. By monotonicity of $\mathsf{U}_{\mathcal{L}}$, $\mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma)) \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$. But $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma))$ since every $T \in \sigma$ is in $\mathcal{L}$ and is its own linearization (if T has pending invocations, we can add any valid response using totality).

Conversely, assume $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$, and let T be a linearizable trace w.r.t. σ. Let T' be an extension of T and $S \in \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau) \subseteq \tau$ such that $T' \rightsquigarrow S$. So we can go from S to T' through a sequence of expansions and commutations, which gives us $T' \in \tau$ by applying axiom (5) of concurrent specifications, and by prefix closure, $T \in \tau$. □

In general, the posets $\mathsf{Spec}_{\mathcal{L}}$ and CSpec ordered by inclusion are not isomorphic, but the above theorem shows that the next best thing one could expect happens: every $\mathcal{L}$-specification has a canonical approximation as a concurrent specification and conversely. Note that Galois connections are widely used for comparing semantics of programs and deriving program analysis methods [24], and are a particular case of adjunctions, which have been promoted as the canonical way of comparing models for concurrency [100].

The right-to-left implication of Theorem 2.32 can be understood as follows: $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ is the smallest concurrent specification that contains σ. Notice that the expansion property (5) is crucial here: without it, we could produce specifications smaller than or incomparable to $\mathsf{Lin}_{\mathcal{L}}(\sigma)$. In fact, the theorem states that $\mathsf{Lin}_{\mathcal{L}}$ is a kind of free construction: starting with the traces in σ, we add all the traces that are required to be in the specification by our axioms, and no other trace than the required ones.

Finally, we have the following factorization lemma:

Lemma 2.33. *Suppose* $\mathcal{L}$ *and* $\mathcal{K}$ *are two sets of traces satisfying the conditions in the beginning of Section 2.2.3, such that* $\mathcal{L} \subseteq \mathcal{K}$*. Then the map* $\mathsf{Lin}_{\mathcal{L}}$ *factors through* $\mathsf{Spec}_{\mathcal{K}}$ *as shown in the following commutative diagram:*

$$\begin{tikzcd}
\mathsf{Spec}_{\mathcal{L}} \arrow[r, "\mathsf{Lin}'_{\mathcal{L}}"] \arrow[rr, bend right, "\mathsf{Lin}_{\mathcal{L}}"'] & \mathsf{Spec}_{\mathcal{K}} \arrow[r, "\mathsf{Lin}_{\mathcal{K}}"] & \mathsf{CSpec}
\end{tikzcd}$$

where $\mathsf{Lin}'_{\mathcal{L}}(\sigma) = \{T \in \mathcal{K} \mid T \text{ is } \mathcal{L}\text{-linearizable with respect to } \sigma\} = \mathsf{Lin}_{\mathcal{L}}(\sigma) \cap \mathcal{K}$.

Proof. To show that $\mathsf{Lin}'_{\mathcal{L}}(\sigma)$ is a $\mathcal{K}$-specification, we go through the proof of Proposition 2.27, and check that the new traces that we construct are still in $\mathcal{K}$. Let $\sigma \in \mathsf{Spec}_{\mathcal{L}}$, we show that $\mathsf{Lin}_{\mathcal{K}} \circ \mathsf{Lin}'_{\mathcal{L}}(\sigma) = \mathsf{Lin}_{\mathcal{L}}(\sigma)$. Let $T \in \mathsf{Lin}_{\mathcal{K}} \circ \mathsf{Lin}'_{\mathcal{L}}(\sigma)$, and observe that if $T \rightsquigarrow S' \rightsquigarrow S$ with $S' \in \mathsf{Lin}'_{\mathcal{L}}(\sigma)$ and $S \in \sigma$, then $T \rightsquigarrow S$. Conversely, suppose $T \rightsquigarrow S$ with $S \in \sigma$. Since $S \in \mathcal{L} \subseteq \mathcal{K}$ and S is its own $\mathcal{L}$-linearization, we have $S \in \mathsf{Lin}'_{\mathcal{L}}(\sigma)$, so we obtain the factorization $T \rightsquigarrow S \rightsquigarrow S$. □

Sequential linearizability

A trace T is *sequential* when the poset $(\mathrm{op}(T), \preceq)$ is totally ordered, we write seq for the set of sequential traces. A *sequential specification* is a seq-specification (i.e., Definition 2.25 with $\mathcal{L} = \mathsf{seq}$). Note that a sequential specification is not a particular case of concurrent specification: it only satisfies the receptivity condition of Definition 2.25, not the stronger one of Definition 2.22. Intuitively, this means a sequential specification does not specify which behaviors are allowed when some of the processes run in parallel. As we mentioned previously, sequential specifications are very well-understood objects: all the usual techniques that can specify sequential data structures, are various ways to describe seq-specifications in the formal sense of Definition 2.25. Sequential linearizability is thus a canonical way to extend a sequential specification to a concurrent one. It is a useful specification technique because it allows us to build a complex mathematical object, $\mathsf{Lin}_{\mathsf{seq}}(\sigma)$, by providing a much more simple description of it, σ.

The notion of seq-linearizability coincides with the usual notion of linearizability, that we call here *sequential linearizability* to avoid confusion. The Galois connection of Theorem 2.32 says that $\mathsf{Lin}_{\mathsf{seq}}(\sigma)$ is the smallest concurrent specification whose set of sequential traces contains σ. Moreover, it is a Galois insertion:

Proposition 2.34. *For every $\sigma \in \mathsf{Spec}_{\mathsf{seq}}$, we have $\mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma)) = \sigma$.*

Proof. The inequality $\sigma \subseteq \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$ is implied by the Galois connection. Let $T \in \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$, i.e., T is both sequential and linearizable w.r.t. σ. Let T' be a completion of T and $S \in \sigma$ such that $T' \rightsquigarrow S$. Since T' is sequential, it is a normal form of the rewriting system (i.e., no rule can be applied): so we must have $T' = S$. Moreover S is required to be in σ, so $T' \in \sigma$ and by prefix-closure, $T \in \sigma$. □

This implies that $\mathsf{Spec}_{\mathsf{seq}}$ is a subposet of CSpec, which justifies calling *linearizable* a concurrent specification in the image of $\mathsf{Lin}_{\mathsf{seq}}$. This is however a strict subposet, as an application of Theorem 2.32:

Proposition 2.35. *There are non-linearizable concurrent specifications.*

Proof. From Proposition 2.34, any linearizable concurrent specification $\tau = \mathsf{Lin}_{\mathsf{seq}}(\sigma)$ must satisfy the equality $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\tau)) = \tau$. Now, consider the concurrent specification SET-COUNT $\subseteq \mathcal{T}$ which is the set of traces whose set of operations has a partition $(E_i)_{i \in I}$ such that every $e, e' \in E_i$ are concurrent and their response value is the cardinal of E_i.

Informally, E_i is a set of pairwise-concurrent processes that "saw" each other. Note that, because of expansion, we cannot require that all processes running in parallel should see each other: the execution (b) is accepted. All other executions (a), (c), (d) and (e) are rejected. In a sequential trace $T \in$ SET-COUNT, every response returns 1. Thus, $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\text{SET-COUNT}))$ only contains traces whose response is 1. But SET-COUNT also has traces with different responses, e.g. $\mathsf{i}_i \cdot \mathsf{i}_j \cdot \mathsf{r}_j^2 \cdot \mathsf{r}_i^2$. Therefore, SET-COUNT is not linearizable. □

Remark 2.36. Here, we are talking about *linearizable specifications*, that is, concurrent specifications that can be expressed using the (standard) linearizability technique. Proposition 2.35 claims that there are objects whose behavior cannot be specified using linearizability. This should not be confused with the somewhat more common terminology of a *linearizable implementation* of an object. Usually, this is used when we have in mind a particular sequential specification σ (say, a list), and we want to implement a concurrent version of it. Saying that the implementation is linearizable simply means that it satisfies the specification $\mathsf{Lin}_{\mathsf{seq}}(\sigma)$.

Set-linearizability

The idea behind set-linearizability [99] is to specify what happens when a set of processes call an object at the same time. In this setting, an execution trace will be a sequence of sets of processes. In each of these sets, all the processes start executing, then all of them must terminate before we proceed with the next set of processes. Set-linearizability was also recently re-discovered by Hemed et al. [60], who call it *concurrency-aware linearizability*. A typical example of a set-linearizable (but not linearizable) object is the *immediate-snapshot* protocol [15] widely used in distributed computability [64]. Another example is Java's *exchanger* that allows two concurrent threads to atomically swap values.

A trace T is *set-sequential* if it is of the form $T = I_1 \cdot R_1 \cdots I_k \cdot R_k$, where each of the I_i is a non-empty sequence of invocations, R_i for $i < k$ is a sequence of responses with the same set of process numbers as I_i, and R_k is a (possibly empty) sequence of responses whose process numbers are included in those of I_k. We write set for the set of such traces. If we consider $\mathcal{L}$-linearizability with $\mathcal{L} = \mathsf{set}$, we recover the previously defined notion of set-linearizability [99]. Of course, Theorem 2.32 still holds, but we do not have an analogue of Proposition 2.34 here: given $\sigma \in \mathsf{Spec}_{\mathsf{set}}$, $\mathsf{U}_{\mathsf{set}}(\mathsf{Lin}_{\mathsf{set}}(\sigma))$ might actually contain more set-sequential traces than σ. This is because set-sequential specifications are not required to satisfy the expansion property nor the commutativity of invocations and of responses. The linearizability map adds just enough set-sequential traces to make these properties verified. A concurrent specification is *set-linearizable* if it is of the form $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ for some $\sigma \in \mathsf{Spec}_{\mathsf{set}}$.

Example 2.37. The SET-COUNT specification defined in the proof of Proposition 2.35 is set-linearizable. It is obtained as $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ where σ is the set of set-sequential traces of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where every response in R_i is returning the value $|I_i|$.

Proposition 2.38. *There are non-set-linearizable concurrent specifications.*

Proof. This is again an application of Theorem 2.32: from the properties of Galois connections, a set-linearizable concurrent specification $\tau = \mathsf{Lin}_{\mathsf{set}}(\sigma)$ must be such that $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau)) = \tau$. Indeed, using the right-to-left implication of Theorem 2.32 we obtain $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau)) \subseteq \tau$. Conversely, since we have $\sigma \subseteq \mathsf{U}_{\mathsf{set}}(\mathsf{Lin}_{\mathsf{set}}(\sigma))$ using the left-to-right implication, and by monotonicity of $\mathsf{Lin}_{\mathsf{set}}$, we get $\tau \subseteq \mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau))$. Define the concurrent specification INTERVAL-COUNT as the set of alternating traces such that every operation e has a response of the form r_i^k, where k is smaller or equal to the number of operations that are overlapping with e. This is a very permissive version of counting: all executions (a)–(e) are allowed. In a set-sequential trace, all return values are at most n, the total number of processes. Therefore, $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau))$ only contains traces whose response is smaller than n. But τ also contains traces with greater responses, as in execution (b) where process P_1 responds 3 even though only two processes are running. We deduce that INTERVAL-COUNT is not set-linearizable. □

Interval-linearizability

Interval-linearizability was introduced in [18] with the aim of going beyond linearizability and set-linearizability, in order to be able to specify every distributed *task*. They prove that every task can be obtained as the restriction to one-shot executions of an interval-linearizable object. We will show that this result actually extends beyond one-shot tasks: every concurrent specification is interval-linearizable. An example of an interval-linearizable (but not set-linearizable) object is the (non-immediate) write-snapshot object.

We write int for the set of all alternating traces. The notions of *interval-sequential specification* and *interval-linearizability* defined in [18] coincide with $\mathcal{L}$-specifications and $\mathcal{L}$-linearizability where $\mathcal{L} = \mathsf{int}$. Interval-sequential specifications are almost the same as concurrent specifications, except that they do not require the expansion property to be satisfied. In fact, it is mentioned in [18] that one can without loss of generality restrict to interval-sequential executions of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where the I_i (resp., R_i) are non-empty sets of invocations (resp., responses). This amounts to enforcing the commutativity of invocations and of responses, which are both consequences of the expansion property. We do not include this assumption here since it will be enforced anyway after we apply linearizability.

$\mathsf{U}_{\mathsf{int}}$ is by definition the "identity" function. As in the case of set-linearizability, an analogue of Proposition 2.34 does not hold, but we have the converse:

Proposition 2.39. *For every* $\tau \in \mathsf{CSpec}$, $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) = \tau$.

Proof. The inclusion $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) \subseteq \tau$ follows from the Galois connection. For the other inclusion, recall that $\mathsf{U}_{\mathsf{int}}$ is the identity, so we want to prove $\tau \subseteq \mathsf{Lin}_{\mathsf{int}}(\tau)$. But this is trivial since every trace is its own linearization. □

We say that a concurrent specification is *interval-linearizable* if it is in the image of $\mathsf{Lin}_{\mathsf{int}}$.

Proposition 2.40. *Every concurrent specification is interval-linearizable.*

Proof. This is an immediate corollary of Proposition 2.39: $\tau = \mathsf{Lin}_{\mathsf{int}}(\tau)$, and τ is (in particular) an interval-sequential specification. □

Thus, putting together Propositions 2.27 and 2.40, we deduce that interval-linearizable objects and concurrent specifications are one and the same. One can see interval-linearizability as a convenient way of defining concurrent specifications in practice. Imagine that we have a concurrent object in mind (say, the COUNT object described informally in the introduction of Section 2.2) and we want to specify its behavior. We choose some set of execution traces σ that we consider to be the correct ones, but we forget to include the execution trace (c); indeed, it looks like a wrong behavior, but as we discussed before, it cannot be avoided because of the expansion property (5). Then interval-linearizability fixes this mistake automatically, by producing the set of execution traces $\mathsf{Lin}_{\mathsf{int}}(\sigma)$ which will include execution (c), as well as any other trace which is required to exist by the expansion property. This is the benefit of interval-linearizability: thanks to it, we do not have to worry about the expansion property, the map $\mathsf{Lin}_{\mathsf{int}}$ does that for us.

The results of the last three sections can be summed up in the following diagram, by factoring $\mathsf{Lin}_{\mathsf{seq}}$ and $\mathsf{Lin}_{\mathsf{set}}$ as follows:

$$\begin{tikzcd}
\mathsf{Spec}_{\mathsf{seq}} \arrow[r, bend left, "\mathsf{Lin}'_{\mathsf{seq}}"] & \mathsf{Spec}_{\mathsf{set}} \arrow[l, bend left, "\mathsf{U}'_{\mathsf{seq}}"] \arrow[r, bend left, "\mathsf{Lin}'_{\mathsf{set}}"] & \mathsf{Spec}_{\mathsf{int}} \arrow[l, bend left, "\mathsf{U}'_{\mathsf{set}}"] \arrow[r, bend left, "\mathsf{Lin}_{\mathsf{int}}"] \arrow[r, phantom, "\perp"] & \mathsf{CSpec} \arrow[l, bend left, "\mathsf{Id}"]
\end{tikzcd}$$

where the $\dashv$ symbol between $\mathsf{Lin}_{\mathsf{int}}$ and Id indicates that they are related by a Galois connection, and the primed functions are the expected ones. Moreover, the two other Galois connections are obtained by composition:

$$\mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} = \mathsf{Lin}_{\mathsf{set}} \dashv \mathsf{U}_{\mathsf{set}} \qquad \text{and} \qquad \mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} \circ \mathsf{Lin}'_{\mathsf{seq}} = \mathsf{Lin}_{\mathsf{seq}} \dashv \mathsf{U}_{\mathsf{seq}}.$$

Other variants

Using the framework of $\mathcal{L}$-linearizability, it is easy to come up with new notions of linearizability, by instantiating $\mathcal{L}$ with another suitable set of traces. For instance, say a trace T is *k-concurrent* if every prefix of T has at most k pending invocations. Intuitively, a trace is k-concurrent if at any time, no more than k processes are running in parallel. Let $\mathcal{L}$ be the set of k-concurrent traces. Then, given a specification that says how an object behaves when it is accessed concurrently by at most k processes, $\mathcal{L}$-linearizability is a canonical way of extending this specification to any number of processes. In their paper about concurrency-aware linearizability [60], Hemed et al. specify Java's exchanger object on traces that are both 2-concurrent and set-sequential, then they apply linearizability to obtain the full concurrent specification.

2.3 A computational model

In Section 2.2, we have defined how to specify the behavior of concurrent objects (Definition 2.22), and we discussed the relevance of this definition by comparing it to other specification techniques. Our goal now is to build a computational model that relies on this notion of concurrent specification.

More precisely, in this section, we provide an operational model for concurrent programs communicating through shared objects. We assume given a set Obj of objects: they might be, for instance, concurrent data structures that have already been implemented, and that our programs are able to use in order to compute and communicate. We do not want to depend on a particular implementation of these objects, but on their specification. Thus, each object comes with its concurrent specification (as in Definition 2.22), which is the set of behaviors that it might exhibit. Note that our model does not have any special construct for reading and writing in the shared memory: we assume that the memory itself is given as an object in Obj, with an appropriate specification. Thus, the only meaningful action a program can take is to call one of the objects; and possibly do some local computation to determine what the next call should be.

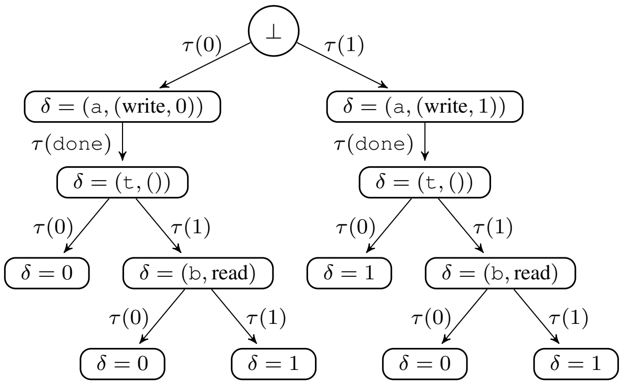
2.3.1 Programs and protocols

To abstract away the syntax of the programming language, we use an automata-like representation, as we did in Section 2.1.3. Morally, our notion of program (Definition 2.41) is the same as the one that we had in the "first approach" section (Definition 2.8). Here, we recall it with updated notations, since we are now working with a different notion of object specification.

The example below shows an implementation of binary consensus among two processes, using three shared objects: two read-write registers `a` and `b`, and a test-and-set object `t`. The pseudo-code written on the left is the program that is run by one of the processes. The other process should run the same program, but where the role of `a` and `b` is reversed. The automaton on the right depicts how this program will be formally represented in our formalism. Each state of the automaton contains the following information:

the current position of the program pointer, as well as the values of the local variables of the program (x and v' in the example), and of the initial input value (v in the example). In a given state of the automaton, the decision function δ indicates which is the next object that the program will call, and which argument is given in the invocation. The transition function τ says what the next state will be depending on the return value of the call.

```
consensus(v) {
  a.write(v);
  x := t.test-and-set();
  if (x == 0)
    return v;
  else
    v' := b.read();
    return v';
}
```



We suppose fixed a set Obj of objects, along with their concurrent specification $\mathsf{spec}(o) \in \mathsf{CSpec}$ for each $o \in \mathsf{Obj}$. Here, a program is basically a piece of code (executed by one of the processes) that takes a value as input, makes several calls to the objects in Obj (using algebraic operations to combine their results) and finally returns a value. Formally,

Definition 2.41. A *program* is a quadruple $(Q, \perp, \delta, \tau)$ consisting of:

- a (possibly infinite) set Q of *local states* containing an *idle state* $\perp \in Q$,
- a *decision function* $\delta : Q \setminus \{\perp\} \to (\mathsf{Obj} \times \mathcal{V}) \cup \mathcal{V}$,
- a *transition function* $\tau : Q \times \mathcal{V} \to Q \setminus \{\perp\}$.

The idle state is the one where the program is waiting to be called with some input value x, in which case it will go to state $\tau(\perp, x)$. After that, the decision function gives the next step of the process depending on the current state: either call some object with a given input value, or terminate and output some value. In the case where an object is called, the transition function gives the new local state of the process, depending on the previous state and the value returned by the object.

Definition 2.42. A *protocol* P is given by a program $P_i = (Q_i, \perp_i, \delta_i, \tau_i)$ for each process $i \in [n]$.

Remark 2.43. It is more common in practice to have all the processes running the same program, since we do not want to write as many programs as there are processes. However, in this context, each process has access to its process number, and can use it to enter parts of the code that the other processes will skip. So, the above presentation with one program for each process is equivalent.

2.3.2 Semantics of a protocol

The *global state* of a protocol P is an element $q = (q_0, \dots, q_{n-1})$ of $\overline{Q} = \prod_i Q_i$, consisting of a state for each process P_i. The initial state is $q_{\text{init}} = (\perp_0, \dots, \perp_{n-1})$ and, given a global state q, we write $q[i \leftarrow q'_i]$ for the state where the i-th component q_i has been replaced by q'_i. We now describe the set $\mathcal{A}$ of possible actions for P, as well as their effect $\Delta : \overline{Q} \times \mathcal{A} \to \overline{Q}$ on global states.

i_i^x: the i-th process is called with input value $x \in \mathcal{V}$. The local state q_i of process i is changed from $\bot_i$ to $\tau_i(\bot_i, x)$:

$$\Delta(q, \mathsf{i}_i^x) \quad = \quad q[i \leftarrow \tau_i(\bot_i, x)]$$

where the state on the right is q where q_i has been replaced by $\tau_i(\bot_i, x)$.

$\mathsf{i}(o)_i^x$: the i-th process invokes the object $o \in \mathsf{Obj}$ with input value $x \in \mathcal{V}$. This does not have any effect on the global state:

$$\Delta(q, \mathsf{i}(o)_i^x) \quad = \quad q$$

$\mathsf{r}(o)_i^x$: the object $o \in \mathsf{Obj}$ returns some output value $x \in \mathcal{V}$ to the i-th process. The local state of process i is updated according to its transition function τ_i:

$$\Delta(q, \mathsf{r}(o)_i^x) \quad = \quad q[i \leftarrow \tau_i(q_i, x)]$$

r_i^x: the i-th process has finished computing, returning the output value $x \in \mathcal{V}$. It goes back to idle state:

$$\Delta(q, \mathsf{r}_i^x) \quad = \quad q[i \leftarrow \bot_i]$$

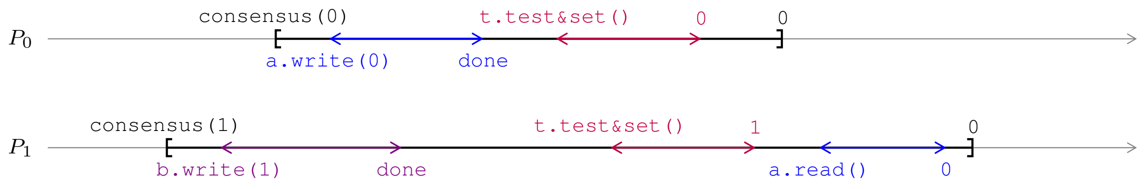
The actions of the form i_i^x and r_i^x (resp. $\mathsf{i}(o)_i^x$ and $\mathsf{r}(o)_i^x$) are called *outer* (resp. *inner*) *actions*. Given a trace $T \in \mathcal{A}^*$ and an object o, we denote by T_o, called the *inner projection on* o, the trace obtained from T by keeping only the inner actions of the form $\mathsf{i}(o)_i^x$ or $\mathsf{r}(o)_i^y$. The function Δ is extended as expected as a function $\Delta : \overline{Q} \times \mathcal{A}^* \to \overline{Q}$, i.e., $\Delta(q, T \cdot T') = \Delta(\Delta(q, T), T')$ and $\Delta(q, \varepsilon) = q$. A trace is valid if at each step in the execution, the next action is taken according to the decision function δ. Formally:

Definition 2.44. A trace $T \in \mathcal{A}^*$ is *valid* when for every strict prefix U of T, writing $T = U \cdot a \cdot V$ and $q = \Delta(q_{\text{init}}, U)$, with $a \in \mathcal{A}$, we have:

- if $a = \mathsf{i}_i^x$ then $q_i = \bot_i$,
- if $a = \mathsf{r}_i^y$ then $q_i \neq \bot_i$ and $\delta_i(q_i) = y$,
- if $a = \mathsf{i}(o)_i^x$ then $q_i \neq \bot_i$ and $\delta_i(q_i) = (o, x)$,
- if $a = \mathsf{r}(o)_i^y$ then $q_i \neq \bot_i$.

Moreover, we require that for every object $o \in \mathsf{Obj}$, the inner projection T_o belongs to $\mathsf{spec}(o)$. The set of valid traces for P is written $\mathcal{T}_P \subseteq \mathcal{A}^*$.

Example 2.45. An example of a valid trace $T \in \mathcal{T}_P$ is depicted below, where the protocol P is the one of Section 2.3.1. The outer actions are represented as black intervals with square brackets. The inner actions are represented as colored intervals with angle brackets. The validity of this trace expresses two things: each process behaves according to its program; and the three objects `a`, `b` and `t` behave according to their specification. Note that being valid does not have anything to do with whether or not the outer actions meet the specification of consensus.



A protocol P is *obstruction-free* if there is no valid infinite trace (i.e., all its finite prefixes are valid) involving only inner-i-actions after some position: in other words, a process running alone will eventually decide on an output value. A protocol P is *wait-free* if there is no valid infinite trace involving infinitely many inner-i-actions and no outer-i-action after some position: in other words, a process will eventually decide on an output, no matter what the other processes do. In particular, a wait-free protocol is also obstruction-free. Given a trace $T \in \mathcal{A}^*$, we write $\pi(T)$ for the trace obtained by keeping only outer actions.

Definition 2.46. The *semantics* of a protocol P is the set of traces $[\![P]\!] = \{\pi(T) \mid T \in \mathcal{T}_P\}$, and P *implements* a concurrent specification σ whenever $[\![P]\!] \subseteq \sigma$, i.e., all the outer traces that P can produce are correct with respect to σ.

Remark 2.47. This notion of implementation might seem surprising at first. Why do we only require $[\![P]\!] \subseteq \sigma$, instead of $[\![P]\!] = \sigma$? The intuition behind the specification σ is that it is the set of all the *correct* execution traces. The elements of σ correspond to behaviors of the object that are *allowed* to happen. But are there cases where we would like to require some behaviors to actually happen?

Most of the time, the inclusion of Definition 2.46 makes sense. For instance, when we talk about solving consensus, what we want is to reach an agreement, we do not care about which value is agreed upon. If two processes start with input values 0 and 1, the output values are allowed to be either 00 or 11. A program that always agrees on 00 in such a situation would be considered correct. Another example from concurrent data structures: as we saw in the previous section, linearizable concurrent lists are the most constrained version of concurrent lists. When we consider weaker implementations of concurrent lists (for example in [57]), the additional traces that we allow are not particularly desirable behaviors. On the contrary, they correspond to "slightly wrong" behaviors that we decide to tolerate, for performance reasons. In particular, a linearizable implementation of lists should also meet these weaker specifications.

However, there are a few cases where this notion of implementation seems to permit trivial solutions. With our COUNT object, as we discussed previously, even when the processes are concurrent, the execution where every process returns 1 cannot be avoided because of the expansion property. So, what if we write a program that always returns 1, without even trying to detect other processes? According to the current definition, this is a good implementation of the COUNT object. A possible way to fix this would be to refine our notions of specification and implementation to have not only a set σ of traces that are *allowed*, but also a set τ of traces that are *required*. Then, a protocol P implements such a specification whenever $\tau \subseteq [\![P]\!] \subseteq \sigma$. This is still not completely satisfactory: we require that some behaviors *can* happen, but we have no idea how often they actually occur. So maybe one would like to equip the set of correct traces with some probability distributions.

While this research direction might be worth pursuing, for the rest of this chapter, we will stick with Definition 2.46. Since our goal is to recover the topological characterization of task solvability, it makes sense to stay as close as possible to Definition 1.29, where we also require an inclusion between carrier maps.

We will now show that if the protocol P is obstruction-free, then $[\![P]\!]$ itself is a concurrent specification (Theorem 2.54). This is expected: no matter what the protocol is, if our notion of concurrent specification is general enough, it should be able to express that the protocol P is doing. Hence, Theorem 2.54 confirms once again that the axioms of Definition 2.22 are well-chosen. Indeed, in our model, the set of traces generated by a protocol necessarily satisfies all of these axioms; and in particular, the expansion property cannot be avoided. In the following, we use uppercase letters T, T' to denote traces containing only outer actions, and lowercase letters w, w', r, s, t to denote traces that might contain both inner and outer actions.

Lemma 2.48. *Every trace $T \in [\![P]\!]$ is alternating.*

Proof. T must begin by an invocation because $q_{\text{init}_i} = \bot_i$ and the only i-action that can occur with local state $\bot_i$ is i_i^x. We then prove by induction on the length of $w \in \mathcal{A}^*$ that $\Delta(q_{\text{init}}, w)_i \neq \bot_i$ if the last outer i-action in w was an invocation, and $\Delta(q_{\text{init}}, w)_i = \bot_i$ if it was a response. This implies that no two invocations nor two responses by the same process can occur consecutively in a valid trace. □

Lemma 2.49. *$[\![P]\!]$ has the expansion property, i.e., for all outer traces T, T', process numbers $i, j \in [n]$ with $i \neq j$, values $x, y \in \mathcal{V}$ and outer action $a_j \in \mathcal{A}$ of process j, we have:*
- *if $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T' \in [\![P]\!]$, then $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T' \in [\![P]\!]$, and*
- *if $T \cdot \mathsf{r}_i^y \cdot a_j \cdot T' \in [\![P]\!]$, then $T \cdot a_j \cdot \mathsf{r}_i^y \cdot T' \in [\![P]\!]$.*

Proof. First, let us give an intuitive explanation of why these two properties hold. For the first one, assume that a valid execution of the protocol P occurred, where process i invoked the object after the action a_j was performed. Then, another execution could have occurred, where instead process i invokes a bit sooner, but it immediately starts idling without performing any inner action. Then, process j cannot distinguish these two executions, so performing the action a_j after the invocation i_i^x is still valid. For the second property, the reasoning is similar, except that in the new execution that we construct, process i idles just before returning its value.

We now formalize these two proofs. Assume $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T' \in [\![P]\!]$, i.e., there is a word $w \in \mathcal{T}_P$ such that $\pi(w) = T \cdot a_j \cdot \mathsf{i}_i^x \cdot T'$. So, w is of the form $t \cdot a_j \cdot s \cdot \mathsf{i}_i^x \cdot t'$ where s does not contain outer actions. We will show that $w' = t \cdot \mathsf{i}_i^x \cdot a_j \cdot s \cdot t' \in \mathcal{T}_P$, and the result follows by projection. Since the inner projections of w and w' are the same, the specifications of the objects are satisfied. We have to show that the conditions regarding the decision functions are respected. Write $q = \Delta(q_{\text{init}}, t)$ and $q' = \Delta(q_{\text{init}}, t \cdot a_j \cdot s)$. Then since w is valid, we have $q'_i = \bot_i$.

Claim 1: the inner trace s does not contain any action of process i. Indeed, only an outer action r_i^y can set the local state of i to $\bot_i$, and s has no outer action: so i's local state is $\bot_i$ during all of s. But inner actions can only be valid when the local state is not $\bot_i$.

Claim 2: Let u be a prefix of s and $q'' = \Delta(q_{\text{init}}, t \cdot a_j \cdot u) = \Delta(q, a_j \cdot u)$. Then $\Delta(q, \mathsf{i}_i^x \cdot a_j \cdot u) = q''[i \leftarrow \tau_i(\bot_i, x)]$. This is proved by induction on the length of the prefix u and using Claim 1.

Claim 3: $\Delta(q, a_j \cdot s \cdot \mathsf{i}_i^x) = \Delta(q, \mathsf{i}_i^x \cdot a_j \cdot s)$. This is proved by taking $u = s$ in Claim 2.

Finally, let u is a prefix of $w' = t \cdot \mathsf{i}_i^x \cdot a_j \cdot s \cdot t'$. If u ends in t or (by Claim 3) in t', the global state after executing it is the same as for w, so the validity condition is verified. If u ends in s, then by Claim 2 the global state only differs by its i component, but since by Claim 1 the next action is not from i, the validity condition is also verified. Therefore $w' \in \mathcal{T}_P$, and hence $\pi(w') = T \cdot \mathsf{i}_i^x \cdot a_j \cdot T' \in [\![P]\!]$.

For the second implication, the situation is very similar: suppose that $w = t \cdot \mathsf{r}_i^y \cdot s \cdot a_j \cdot t' \in \mathcal{T}_P$, and show that $w' = t \cdot s \cdot a_j \cdot \mathsf{r}_i^y \cdot t' \in \mathcal{T}_P$. Once again, we can show that s contains only inner actions, and none of them is from process i. The analogue of Claim 3 says that $\Delta(q_{\text{init}}, t \cdot \mathsf{r}_i^y \cdot s \cdot a_j) = \Delta(q_{\text{init}}, t \cdot s \cdot a_j \cdot \mathsf{r}_i^y)$, and it is also proved by first generalizing it to prefixes of s. □

Remark 2.50. The key reason that makes Lemma 2.49 go through is the fact that the actions i_i^x and r_i^y do not have any effect besides starting a process or terminating it. Taking these steps does not communicate any information to the other processes. For example, a process might start running and then wait for a while before doing any "real" computation. Such a process cannot be seen by the others. This fact is an arbitrary choice in how we designed our computational model, but it reflects what really happens in practice: calling a function, or returning a value, both consume some amount of clock cycles from the processor, and they do not usually have any effect on the shared memory. Thus, one could possibly have a process that has started running, but was immediately preempted by the scheduler before it could perform any meaningful operation.

Lemma 2.51. *$[\![P]\!]$ is closed under prefix.*

Proof. $\mathcal{T}_P$ is closed under prefix since for all $o \in \mathsf{Obj}$, $\mathsf{spec}(o)$ is closed under prefix by definition, and the conditions on the decision functions are also preserved. Then, $[\![P]\!]$ is also closed under prefix. □

Lemma 2.52. *$[\![P]\!]$ is receptive.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ does not have a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is a response, and thus we have $\tau(q_{\text{init}}, w)_i = \bot_i$ (cf. proof of Lemma 2.48). So $w \cdot \mathsf{i}_i^x$ is valid for all x, and $T \cdot \mathsf{i}_i^x \in [\![P]\!]$. □

Lemma 2.53. *$[\![P]\!]$ is total.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ has a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is an invocation, so the local state of i after executing w is $q_i \neq \bot_i$. If $\delta_i(q_i) = y \in \mathcal{V}$, then $w \cdot \mathsf{r}_i^y$ is a valid execution, which concludes the proof. Otherwise, $\delta_i(q_i) = (o, x)$. Then $w \cdot \mathsf{i}(o)_i^x$ is valid because the object o is receptive. And since o is total, there exists y such that $w \cdot \mathsf{i}(o)_i^x \cdot \mathsf{r}(o)_i^y$ is valid. The new local state of i after executing this trace is $q_i' \neq \bot_i$, so we can iterate the previous reasoning. Eventually, we will reach some local state q_i'' with $\delta_i(q_i'') = y'' \in \mathcal{V}$, because P is obstruction-free (i.e., there is no infinite execution ending with only inner i-actions). □

Putting all the previous lemmas together, we obtain Theorem 2.54. Notice that the obstruction-free assumption was only used to prove totality; all the other axioms are true for any program.

Theorem 2.54. *The semantics $[\![P]\!]$ of an obstruction-free protocol is a concurrent specification.*

This theorem ensures that the axioms of concurrent specifications are reasonable, in the sense that they are validated in our model. Thus, our concurrent specifications are the only ones of interest: specifications that do not satisfy these axioms cannot be implemented. For instance, this theorem implies that any protocol implementing a COUNT-like object *must* accept execution (c), since it is obtained by expanding a valid sequential execution, and the protocol's behavior is closed under expansion.

2.4 From trace semantics to geometric semantics

In Sections 2.2 and 2.3, we have defined a general notion of concurrent object specifications, and a computational model where several processes can communicate through these objects. The purpose of these two sections was achieved at Definition 2.46: we now have a concrete meaning of what is means to *implement* an object. Our goal now is to compare this notion of implementation to the topological notion of *solving* a concurrent task (Definition 1.29). By doing so, we will obtain an analogue of Herlihy and Shavit's *Asynchronous Computability Theorem* [70] for arbitrary objects (Theorem 2.73), which is the main goal of this chapter.

This section is organized as follows. First, in Section 2.4.1, we study more in-depth the relationship between tasks ans objects. Then in Section 2.4.3 we show how we can associate a protocol complex to any protocol, in a systematic way. Finally, our generalized version of the asynchronous computability theorem is proved in Section 2.4.4.

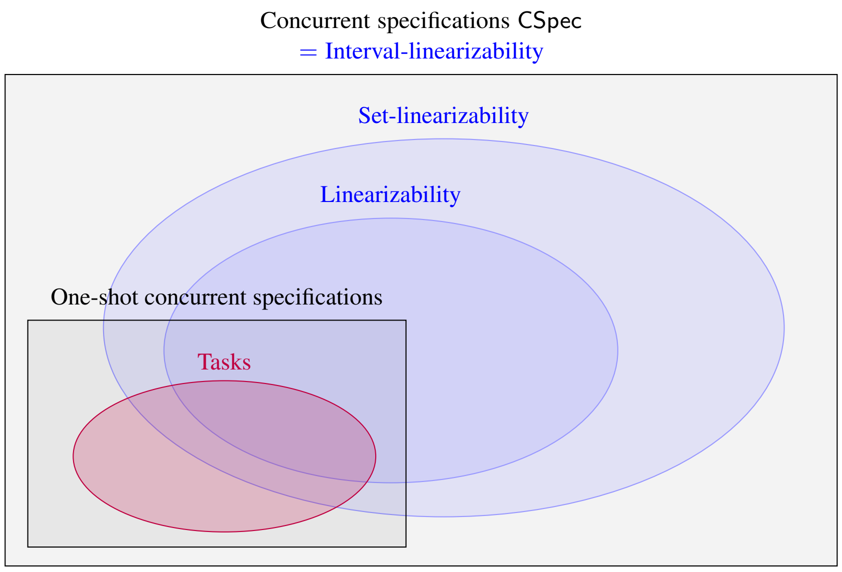
2.4.1 Tasks as one-shot objects

The topological approach to fault-tolerant distributed computing [64] is not interested in implementing long-lived objects as in the previous sections, but in solving *decision tasks*. We already discussed the differences and similarities between objects and tasks, in Section 2.2.1 and throughout the chapter. For the sake of clarity, we briefly recall the key points that will matter in this section.

1. Tasks might exhibit complex concurrent behavior, whereas many common objects (e.g., the linearizable ones) usually behave as in a sequential setting.
2. Concurrent objects are *long-lived* (i.e., each process can call the object several times), whereas tasks are *one-shot* (once a process decides on an output, it does not participate in the computation again).

The first point was tackled by Castañeda et al. in [18], where they defined the notion of interval-linearizability that subsumes both tasks and objects. Since we showed in Propositions 2.27 and 2.40 that our notion of concurrent specification (Definition 2.22) coincides with interval-linearizable specifications, the same result holds for CSpec. Thus, tasks are a special case of objects; and by point 2, they are even a subset of *one-shot concurrent specifications*, that is, objects where only the first call of each process matters.

A more surprising fact, also noted in [18], is that tasks are actually weaker than one-shot objects: there exists one-shot objects that cannot be expressed as tasks. The situation is summed up in the diagram below. To fix this issue, they defined a notion of "refined task", which is as expressive as one-shot objects. Our goal here is dual: we want to characterize the subclass of one-shot objects that correspond to tasks.


Concurrent specifications CSpec
= Interval-linearizability
Set-linearizability
Linearizability
One-shot concurrent specifications
Tasks

Let us start by defining formally what a one-shot concurrent object specification is. Intuitively, this is very close to Definition 2.22, except that we are only interested in the execution traces where each process calls the object at most once. We say that a trace $T \in \mathcal{T}$ is *one-shot* if for every $i \in [n]$, $\pi_i(T)$ contains at most one invocation and one response; that is, $\pi_i(T)$ can be either ε or i_i^x or $\mathsf{i}_i^x \mathsf{r}_i^y$ for some values $x, y \in \mathcal{V}$. We write $\mathcal{T}_1$ for the set of all one-shot traces.

Definition 2.55. A *one-shot concurrent specification* σ is a subset of $\mathcal{T}_1$ which is

(1) *prefix-closed*: if $T \cdot T' \in \sigma$ then $T \in \sigma$,

(2) *non-empty*: $\varepsilon \in \sigma$,

(3”) *receptive among one-shot traces*: if $T \in \sigma$ and $\pi_i(T) = \varepsilon$, then $T \cdot \mathsf{i}_i^x \in \sigma$ for every $x \in \mathcal{V}$,

(4) *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$,

(5) is closed under *expansion*:

- if $T \cdot a_j \cdot \mathsf{i}_i^x \cdot T' \in \sigma$ where a_j is an action of process $j \neq i$, then $T \cdot \mathsf{i}_i^x \cdot a_j \cdot T' \in \sigma$.
- if $T \cdot \mathsf{r}_i^y \cdot a_j \cdot T' \in \sigma$ where a_j is an action of process $j \neq i$, then $T \cdot a_j \cdot \mathsf{r}_i^y \cdot T' \in \sigma$.

We write CSpec_1 for the set of one-shot concurrent specifications.

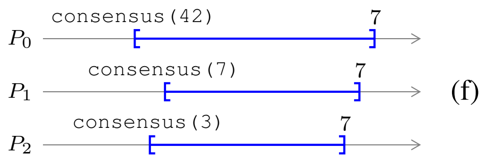
The only difference with Definition 2.22 is the receptivity axiom (3”), which only requires an object to accept the very first invocation of each process. The behavior of the object for any subsequent calls is not specified.

Remark 2.56. Note that with this definition, a one-shot concurrent specification σ is *not* a concurrent specification. The reason for this is that a one-shot specification contains only one-shot traces, whereas axiom (3) of Definition 2.22 implies that a concurrent specification must contain traces that are not one-shot. So, formally, CSpec_1 is not a subset of CSpec.

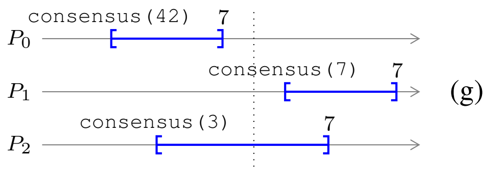
However, as depicted in the Venn diagram above, we will often refer to one-shot specifications as a particular case of concurrent specifications. The reason for this is that there is a canonical way to turn a one-shot specification into a regular one, by saying that, whenever a process tries to invoke the object twice, the response to any of the subsequent calls can have any output value. Indeed, since the one-shot specification does not say how the object behaves on traces that are not one-shot, we should not impose any restrictions on them. More formally, there is an injection $\iota : \mathsf{CSpec}_1 \to \mathsf{CSpec}$ defined as $\iota(\sigma) = \{T \in \mathcal{T} \mid \text{every one-shot prefix of } T \text{ is in } \sigma\}$. In the remainder of the section, we keep this injection implicit.

Example 2.57 (One-shot linearizable list). An object which is not very useful in practice, but easy to describe, is the one-shot version of a concurrent list. It behaves just like a list, with two operations `push` and `pop`, except that each process is allowed to use it only once (either to push a value, or to pop a value, but not both). To define it formally, first take $\sigma_{\text{list}} \in \mathsf{Spec}_{\mathsf{seq}}$, the sequential specification of a list. It can be described by usual sequential techniques, e.g., an automaton as in Definition 2.1. Then, $\mathsf{Lin}_{\mathsf{seq}}(\sigma_{\text{list}}) \in \mathsf{CSpec}$ is the usual concurrent specification of a linearizable list. To get a one-shot version of it, we simply remove all the traces that are not one-shot: we obtain the one-shot specification $\sigma_{\text{os-list}} \in \mathsf{CSpec}_1$, defined as $\sigma_{\text{os-list}} = \{T \in \mathsf{Lin}_{\mathsf{seq}}(\sigma_{\text{list}}) \mid T \text{ is one-shot}\}$.

Example 2.58 (Consensus object). A more natural example is of course the consensus object. Each process can propose an input value k by calling the only method `consensus(k)` of the object. The return value should be one of the proposed values; so that every process receives the same output. Even if this object is well-known when described as a task, it can be a bit tricky to formally define the corresponding set of one-shot traces. It is clear which fully-concurrent traces should be accepted; for example, the trace (f) below is obviously correct.



(f)



(g)

But what about trace (g)? If we just look at the sets of input and output values, they respect the task specification. However, remember that one-shot specifications must be prefix-closed. If we cut the trace before the vertical dotted line, process P_0 is returning a value that has not been proposed yet. So, the execution trace (g) should actually not be considered correct. We will explain below how to properly define the consensus one-shot object.

We now give a formal definition of tasks. The formalism that we use here is similar to the one used in Herlihy and Shavit's paper where the asynchronous computability theorem was first introduced [70]. Although it does not explicitly refer to simplicial complexes, it is a direct reformulation of the usual topological definition of a task (Definition 1.19). Recall that the set of values is written $\mathcal{V}$ and n is the number of processes. Let $\perp$ be a fresh symbol which is not in $\mathcal{V}$. A *vector* is a tuple $U \in (\mathcal{V} \cup \{\perp\})^n$, such that at least one component of U is not $\perp$. We write U_i for the i-th component of U, and $U[i \leftarrow x]$ for the vector U where the i-th component U_i has been replaced by $x \in \mathcal{V}$. Given two vectors U and V, we say that U *matches* V when $U_i = \perp$ iff $V_i = \perp$. We say that U is a *face* of V, written $U \preceq V$, when for all i, either $U_i = V_i$ or $U_i = \perp$. A vector U is *maximal* if none of its components is $\perp$. If X is a set of vectors, its *downward closure* is $\downarrow X = \{U \mid U \preceq V \text{ for some } V \in X\}$.

Definition 2.59. A *task* is a triple $\Theta = (I, O, \Delta)$ such that:

– $I = \mathcal{V}^n$ and $O \subseteq \mathcal{V}^n$ are sets of maximal vectors. The elements of I (resp. of O) and their faces are called *input* vectors (resp. *output* vectors).
– $\Delta \subseteq \downarrow I \times \downarrow O$ is a relation between input and output vectors, such that:
  - Δ only relates matching vectors,
  - for every input vector $U \in \downarrow I$, $\Delta(U) \neq \varnothing$,
  - for all input vectors $U \preceq U'$, $\Delta(U) \subseteq \downarrow \Delta(U')$.

Above, we write $\Delta(U) = \{V \mid (U, V) \in \Delta\}$. Intuitively, the task specification says that if each process i starts with the input U_i, the set $\Delta(U)$ consists of all the output vectors that are considered acceptable outputs. A '$\perp$' component in a vector represents a process that is not participating in the computation, either because it crashed, or because it was so slow that all the other processes terminated before it could take any steps. With that in mind, the third condition on Δ asserts that, if such a "slow" process wakes up, there is at least one valid output that is compatible with what the other processes already decided.

Remark 2.60. Note that there is a little mismatch between this definition and the usual definition of a task in distributed computing: namely, we are requiring $I = \mathcal{V}^n$, whereas the more usual definition would allow any $I \subseteq \mathcal{V}^n$. The reason why we always require a full input complex is that we want to match the receptivity property of concurrent specifications, where an object must always accept any input.

Intuitively, with our definition, a task *must* specify, for each input vector, what the corresponding outputs should be. This is not a restriction compared to the usual definition of a task: if we do not care about what happens on some of the input vectors, we can simply consider that every output is correct. This is the same idea as in Remark 2.56: unspecified behavior is equivalent to saying that every output is correct.

We now describe how we can turn a task into a one-shot object. Given a vector U, the set $\{i \in [n] \mid U_i \neq \bot\}$ is called the *participating set* of U. A pair $(U, V) \in \Delta$, where U and V have participating set $I = \{i_1, \ldots, i_k\}$, corresponds to a correct execution trace of the form $\mathsf{i}_{i_1}^{U_{i_1}} \cdots \mathsf{i}_{i_k}^{U_{i_k}} \cdot \mathsf{r}_{i_1}^{V_{i_1}} \cdots \mathsf{r}_{i_k}^{V_{i_k}}$. This is a fully-concurrent trace, that is, a trace where all the participating processes invoke with the input values from U, and then they all respond with the output values from V. Remember that the order of consecutive invocations or consecutive responses does not matter. For convenience, we will write such a trace $\mathsf{i}^U \cdot \mathsf{r}^V$.

Thus, the task specification Δ gives us a set of execution traces $\{\mathsf{i}^U \cdot \mathsf{r}^V \mid (U, V) \in \Delta\} \subseteq \mathcal{T}$. But this set does not satisfy the conditions of Definition 2.22: we need to specify which traces are correct or not, among the traces of other "shapes" (i.e., traces where invocations and responses are interleaved). For example, consider a trace of the form $\mathsf{i}^U \cdot \mathsf{r}^V \cdot \mathsf{i}_j^x \cdot \mathsf{r}_j^y$, where $(U, V) \in \Delta$ have a participating set I, and $j \notin I$. Intuitively, this represents the situation where all the processes in I ran together without hearing from j, and after all of these processes terminated, the process j ran alone with input x and decided output y. What should be the condition on x and y for this trace to be considered correct? The most sensible answer is that we should require $(U[j \leftarrow x], V[j \leftarrow y]) \in \Delta$. The next definition generalizes this idea to traces of any shape.

Definition 2.61. Let $\Theta = (I, O, \Delta)$ be a task. We define the one-shot concurrent specification $G(\Theta) \subseteq \mathcal{T}_1$ as the set of execution traces $T \in \mathcal{T}$ such that:

– T is one-shot, i.e., every process has at most one invocation and one response in T,
– for every prefix S of T, there exists a completion S' of S (obtained by appending responses to the pending invocations), such that $(U_{S'}, V_{S'}) \in \Delta$, where the i-th component of $U_{S'}$ (resp. $V_{S'}$) is x if i_i^x (resp., r_i^x) appears in S, and $\bot$ otherwise.

Example 2.62 (Consensus object). With Definition 2.61, we can now define precisely the consensus object mentioned in Example 2.58. Let Θ be the usual specification of the consensus task; then $G(\Theta)$ is the set of traces allowing us to view it as a one-shot object specification. In particular, $G(\Theta)$ does not contain execution (g). Indeed, if we take the prefix S corresponding to the part before the dotted line, there is no way to complete it (by adding a response to process P_2) so that the task specification is satisfied.

Proposition 2.63. *$G(\Theta)$ is a one-shot concurrent specification.*

Proof. Prefix-closure (1) and non-emptiness (2) are obvious.

– (3"): Let $T \in G(\Theta)$ and assume that no action from process i occurs in T. Let $x \in \mathcal{V}$, we want to show that $T \cdot \mathsf{i}_i^x \in G(\Theta)$. It is a one-shot trace, since T is one-shot and does not contain actions from process i. Let S be a prefix of $T \cdot \mathsf{i}_i^x$. If it is a strict prefix, then it is also a prefix of T, and since $T \in G(\Theta)$ we are done. The only remaining prefix is the trace $S = T \cdot \mathsf{i}_i^x$ itself.

  What we have to do is find responses to the pending invocations of S. They are either pending invocations of T, or the last invocation i_i^x. Since T is a prefix of itself and $T \in G(\Theta)$, we know that there is a completion T' of T, such that $(U_{T'}, V_{T'}) \in \Delta$. Write $T' = T \cdot \widehat{T}$, so that $\widehat{T}$ contains the responses to the pending invocations of T.

Now, we want to find a response to the invocation i_i^x that obeys the specification of the task Θ. Since i does not participate in T, the vector $U_{T'}$ has a $\perp$ at position i, and so $U_{T'} \preceq U_{T'}[i \leftarrow x]$. Thus, we have $\Delta(U_{T'}) \subseteq \downarrow\Delta(U_{T'}[i \leftarrow x])$, since Θ is a task. So in particular $V_{T'} \in \downarrow\Delta(U_{T'}[i \leftarrow x])$, which means that there is some $W \in \Delta(U_{T'}[i \leftarrow x])$ which extends $V_{T'}$. Let $y \in \mathcal{V}$ be the i-th component of W. Pick the complete trace $S' = T \cdot \mathsf{i}_i^x \cdot \widehat{T} \cdot \mathsf{r}_i^y$. Then we can check that $(U_{S'}, V_{S'}) = (U_{T'}[i \leftarrow x], W) \in \Delta$, and therefore $T \cdot \mathsf{i}_i^x \in G(\Theta)$.

– (4): Let $T \in G(\Theta)$ and assume that $\pi_i(T)$ has a pending invocation. Using the same notations as before, the suffix $\widehat{T}$ that completes T must contain a response r_i^y to the pending invocation of process i. We need to prove that $T \cdot \mathsf{r}_i^y \in G(\Theta)$. As before, it is one-shot and all its strict prefixes satisfy the required condition. For $S = T \cdot \mathsf{r}_i^y$, up to a (harmless) reordering of the responses in $\widehat{T}$, we can just take the same completion $S' = T \cdot \widehat{T}$, which gives $T \cdot \mathsf{r}_i^y \in G(\Theta)$.

– (5): Let $T = T_1 \cdot a_j \cdot \mathsf{i}_i^x \cdot T_2 \in G(\Theta)$, with $j \neq i$; we want to prove that $T' = T_1 \cdot \mathsf{i}_i^x \cdot a_j \cdot T_2 \in G(\Theta)$. Let S be a prefix of T'. The only case that could fail is if we split between i_i^x and a_j: if we split before, S is also a prefix of T and has the required property; if we split after, we can split T at the same place and use the same suffix to complete the trace. So the last remaining case is if $S = T_1 \cdot \mathsf{i}_i^x$, but we already did it while proving property (3") above.

For the second half of condition (5), assume $T = T_1 \cdot \mathsf{r}_i^y \cdot a_j \cdot T_2 \in G(\Theta)$ with $j \neq i$, and let us prove that $T' = T_1 \cdot a_j \cdot \mathsf{r}_i^y \cdot T_2 \in G(\Theta)$. As is the other case, the only interesting prefix that we need to complete is $S = T_1 \cdot a_j$. Note that in this trace, there is a pending invocation from process i. Let $\widehat{T}$ be the completion of the trace $T_1 \cdot \mathsf{r}_i^y \cdot a_j$ that is provided by the fact that $T \in G(\Theta)$. We can then complete S by taking $S' = T_1 \cdot a_j \cdot \mathsf{r}_i^y \cdot \widehat{T}$; and we easily check that $(U_{S'}, V_{S'}) \in \Delta$. □

So we have a canonical way to turn a task into a one-shot specification. There is also a map in the other direction: from a one-shot concurrent specification σ, we can produce a task $F(\sigma)$. This direction is much easier to define, all we have to do is keep all the traces of σ that consist of a sequence of invocations followed by a sequence of responses, and put in Δ the corresponding pair (U, V).

Definition 2.64. Given a one-shot concurrent specification $\sigma \subseteq \mathcal{T}$, the task $F(\sigma) = (I, O, \Delta)$ is defined as follows. For each trace $T \in \sigma$ of the form $T = \mathsf{i}_{i_1}^{x_1} \cdots \mathsf{i}_{i_k}^{x_k} \cdot \mathsf{r}_{i_1}^{y_1} \cdots \mathsf{r}_{i_k}^{y_k}$ (we say that T is *fully-concurrent*), we define the vectors U_T (resp., V_T) whose i_j-th component is x_j (resp., y_j) and all the other components are $\perp$. Then $\Delta = \{(U_T, V_T) \mid T \in \sigma \text{ is fully-concurrent}\}$, and I (resp., O) is the set of all the maximal input (resp., output) vectors that appear in Δ.

Proposition 2.65. *$F(\sigma)$ is a task.*

Proof. First, we justify that $\Delta \subseteq \downarrow I \times \downarrow O$. Let $(U_T, V_T) \in \Delta$ for some trace T. We want to show that U_T and V_T are faces of maximal vectors that appear in Δ. We will find a fully-concurrent trace $T' \in \sigma$ such that $U_T \preceq U_{T'}$ and $V_T \preceq V_{T'}$. For each process i that does not participate in T, by the receptivity property of σ, we can add an invocation i_i^x at the end of T. By totality, this invocation has an appropriate response r_i^y, that we add at the end of the trace. Then by applying the expansion property several times, we can push all the new invocations to the left to obtain the trace $T' \in \sigma$, which is fully-concurrent.

We then check that the three conditions on Δ are satisfied. Δ always relates matching vectors because the trace T has matching invocation and responses. For any input vector U, we can use totality of σ to find matching responses, which shows that $\Delta(U) \neq \varnothing$. Finally, given two input vectors $U \preceq U'$, and an element V of $\Delta(U)$, let T be the trace witnessing that $(U, V) \in \Delta$. Then, as in the first paragraph, we can add the missing invocations by receptivity, and matching responses by totality, and get a fully-concurrent trace by expansion. This yields a pair $(U', V') \in \Delta$, with $V \preceq V'$, which is what we want. $\square$

The maps F and G are not inverse of each-other: as we stated in the introduction of the section, one-shot objects are more expressive than tasks (an example of a simple object that cannot be expressed as a task is exhibited in [18]). The next Theorem shows that we still have a close correspondence between tasks and objects: F and G form a Galois connection, as represented in the following diagram.

$$\begin{tikzcd} \mathsf{CSpec}_1 \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] & \mathsf{Tasks} \arrow[l, bend left, "G"] \end{tikzcd}$$

Given two tasks $\Theta = (I, O, \Delta)$ and $\Theta' = (I', O', \Delta')$, we write $\Theta \subseteq \Theta'$ when $\Delta \subseteq \Delta'$.

Theorem 2.66. *The maps F and G are monotonic, and they form a Galois connection between tasks and one-shot objects: for every task Θ and one-shot specification σ, we have*

$$\sigma \subseteq G(\Theta) \iff F(\sigma) \subseteq \Theta$$

Moreover, the following equality holds: $F \circ G(\Theta) = \Theta$.

Proof. The monotonicity of F and G follows directly from the definitions. In the rest of the proof, we write $F(\sigma) = (I', O', \Delta')$.

- ($\Rightarrow$): Assume $\sigma \subseteq G(\Theta)$. Let $(U_T, V_T) \in \Delta'$, for some fully-concurrent trace $T \in \sigma$. So, we also have $T \in G(\Theta)$. Since T is already complete (and a prefix of itself), we obtain $(U_T, V_T) \in \Delta$.
- ($\Leftarrow$): Conversely, assume $F(\sigma) \subseteq \Theta$, and let $T \in \sigma$. Let S be a prefix of T. By prefix-closure, $S \in \sigma$, and by totality, we can add responses to the pending invocations of S and get a complete trace $S' \in \sigma$. Then, using the expansion property, we can push all the invocations to the left in order to get a trace $S'' \in \sigma$, which is fully-concurrent. Moreover, Since S'' is obtained by reordering the actions of S', we have $U_{S'} = U_{S''}$ and $V_{S'} = V_{S''}$. By definition of $F(\sigma)$, we have $(U_{S''}, V_{S''}) \in F(\sigma)$, so by assumption $(U_{S'}, V_{S'}) = (U_{S''}, V_{S''}) \in \Delta$.
- From the first implication, since $G(\Theta) \subseteq G(\Theta)$, we get $F \circ G(\Theta) \subseteq \Theta$. To prove the other inclusion, take $(U, V) \in \Delta$, and consider the associated trace $T = \mathsf{i}^U \cdot \mathsf{r}^V$. We have (by construction) $U = U_T$ and $V = V_T$, so we just need to check that $T \in G(\Theta)$. So let S be a prefix of T; we can choose T itself as the completion of S, and by assumption we have $(U_T, V_T) \in \Delta$. $\square$

Theorem 2.66 tells us a lot about the relationship between tasks and one-shot objects. The implication from left to right explains in what sense G is a canonical way to turn a task into a one-shot object: $G(\Theta)$ is the largest one-shot specification whose set of fully-concurrent traces obeys the task Θ. The fact that $G(\Theta)$ is the largest such specification means that it is the *least restrictive* one; i.e., we did not impose any other conditions beyond what the task Θ requires. Moreover, the equality $F \circ G = \mathsf{id}$ tells us that G is an injection of tasks into one-shot objects. Thus, the objects that we are interested in are precisely the ones in the image of G.

Definition 2.67. A *task-object* σ is a one-shot concurrent specification that can be written as $\sigma = G(\Theta)$ for some task Θ.

Remark 2.68. There exist one-shot concurrent specifications that are not task-objects. An example of such an object is the one-shot list of Example 2.57. A detailed proof of this fact can be found in [18], where it was first remarked.

The maps F and G form a bijection between tasks and task-objects: indeed, given a task-object $\sigma = G(\Theta)$, we have $G \circ F(\sigma) = G \circ F \circ G(\Theta) = G(\Theta) = \sigma$. Conversely, if a one-shot object σ satisfies $\sigma = G \circ F(\sigma)$, then it is a task-object (corresponding to the task $F(\sigma)$). Thus, we have a characterization of task-objects, which does not refer to the notion of task: σ is a task-object *iff* $G \circ F(\sigma) = \sigma$. In fact, since the inclusion $\sigma \subseteq G \circ F(\sigma)$ holds for any one-shot object (as a consequence of Theorem 2.66), we even have the equivalence: σ is a task-object *iff* $G \circ F(\sigma) \subseteq \sigma$. If we reformulate this inclusion by unfolding the definitions of F and G, we obtain a kind of closure property, in the style of Definition 2.55:

(6) *Task property:* for every one-shot trace $T \in \mathcal{T}$, if every prefix S of T can be completed to a trace S' such that expanding S' gives a fully-concurrent trace $S'' \in \sigma$, then $T \in \sigma$.

Then, a *task-object* is a set of one-shot traces that satisfies the axioms $(1) - (6)$.

Now that we have clearly identified the notion of task-object, we will be able to formulate our version of the asynchronous computability theorem: it says that *implementing* a task-object, the sense of Definition 2.46, is the same as *solving* the corresponding task, in the sense of Definition 1.29.

2.4.2 Simplicial tasks

As we mentioned earlier, the notion of task of Definition 2.59 is a direct reformulation of the usual definition that is used in the context of combinatorial topology. For the sake of completeness, we now recall the usual definition, and explain briefly why it is the same as Definition 2.59. Recall that the set of values is $\mathcal{V}$, and n is the number of processes. Moreover, when we talk about chromatic complexes, the underlying set of colors will be $[n]$.

Definition 2.69. A *simplicial task* is a triple $(\mathcal{I}, \mathcal{O}, \Xi)$, where:

- $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I})$ is the pure chromatic simplicial complex of dimension $(n-1)$, whose vertices are of the form (i, v) for all $i \in [n]$ and $v \in \mathcal{V}$, and which contains all the simplices that are well-colored. $\mathcal{I}$ is called the *input complex*.
- $\mathcal{O} = (V_\mathcal{O}, S_\mathcal{O}, \chi_\mathcal{O}, \ell_\mathcal{O})$ is a pure chromatic simplicial complex of dimension $(n-1)$, together with a labeling $\ell_\mathcal{O} : V_\mathcal{O} \to \mathcal{V}$, such that every vertex is uniquely identified by its color and its label. $\mathcal{O}$ is called the *output complex*.
- $\Xi : \mathcal{I} \to 2^\mathcal{O}$ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{O}$.

Note that there is a slight difference with Definition 1.19: as in Definition 2.59, we force the input complex to contain every possible combination of input values, which is a bit unusual but can safely be assumed without loss of generality (see Remark 2.60). There is a straightforward bijection between tasks and simplicial tasks. Consider a task $\Theta = (I, O, \Delta)$ in the sense of Definition 2.59. A vector $U \in (\mathcal{V} \cup \{\bot\})^n$ corresponds to the simplex $X = \{(i, U_i) \mid i \in [n] \text{ and } U_i \neq \bot\}$, where the vertex (i, U_i) is colored by i and labeled by U_i. Two matching vectors correspond to simplices with the same dimension and set of colors. A maximal vector (i.e., with no $\bot$ component) corresponds to a simplex of dimension $(n-1)$. Thus, the sets of maximal vectors I and O of a task correspond to the facets of the corresponding pure simplicial complexes $\mathcal{I}$ and $\mathcal{O}$; and their downward closures $\downarrow I$ and $\downarrow O$ correspond to $\mathcal{I}$ and $\mathcal{O}$. The fact that the relation $\Delta \subseteq \downarrow I \times \downarrow O$ relates only matching vectors, along with the non-emptiness of $\Delta(U)$, is equivalent to saying that the corresponding carrier map is rigid and chromatic. The last remaining condition is monotonicity of Δ, which must also hold for carrier maps. With that correspondence in mind, in the rest of the section, we will not distinguish tasks and simplicial tasks.

2.4.3 The protocol complex

As in the previous section, we recall the usual topological definition of a protocol. It is very similar to Definition 1.25, except that we once again require the input complex to contain every possible combination of input values.

Definition 2.70. A *simplicial protocol* is a triple $(\mathcal{I}, \mathcal{P}, \Psi)$, where:

- $\mathcal{I} = (V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I})$ is the pure chromatic simplicial complex of dimension $(n-1)$, whose vertices are of the form (i, v) for all $i \in [n]$ and $v \in \mathcal{V}$, and which contains all the simplices that are well-colored. $\mathcal{I}$ is called the *input complex*.
- $\mathcal{P} = (V_\mathcal{P}, S_\mathcal{P}, \chi_\mathcal{P}, \ell_\mathcal{P})$ is a pure chromatic simplicial complex of dimension $(n-1)$, together with a labeling $\ell_\mathcal{P} : V_\mathcal{P} \to \mathsf{Views}$, where Views is an arbitrary set of *views*, such that every vertex is uniquely identified by its color and its label. $\mathcal{P}$ is called the *protocol complex*.
- $\Psi : \mathcal{I} \to 2^\mathcal{P}$ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{P}$.

Remember that the intended meaning of Definitions 2.69 and 2.70 is the following: the simplicial protocol $(\mathcal{I}, \mathcal{P}, \Psi)$ *solves* the simplicial task $(\mathcal{I}, \mathcal{O}, \Xi)$ if there exists a chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Psi$ is carried by Ξ. In [64], this is taken as the definition of what it means for a protocol to solve a task. In Section 2.3.1, we have given more concrete definitions of protocols and solvability. Our goal in this section is the following: given a concrete protocol P (as in Definition 2.42), define a simplicial protocol $(\mathcal{I}, \mathcal{P}, \Psi)$, so that the concrete definition of solvability (Definition 2.46) will agree with the simplicial one.

Remark 2.71. In the previous section, we said that tasks and simplicial tasks are a straightforward reformulation of one another. This is not at all the case for protocols and simplicial protocols: they contain very different information. A protocol is comprised of all the programs that the processes run in parallel; it allows us to reconstruct the possible executions of the programs. On the other hand, a simplicial protocol simply contains the local views of each process at the end of each execution; but it says nothing about the computations that occurred in order to produce these views. Defining correctly the protocol complex $\mathcal{P}$ associated to a concrete protocol P is a crucial step in proving our generalized asynchronous computability theorem.

Let Obj be the set of objects that our programs are allowed to use. Let $P = (P_i)_{i \in [n]}$ be a wait-free protocol in the sense of Definition 2.42. Recall that each $P_i = (Q_i, \bot_i, \delta_i, \tau_i)$ is the program of process i. As before, we write $\mathcal{A} = \{\mathsf{i}_i^x, \mathsf{r}_i^x, \mathsf{i}(o)_i^x, \mathsf{r}(o)_i^x \mid o \in \mathsf{Obj}, i \in [n], x \in \mathcal{V}\}$ for the set of actions of the protocol. The effect of a trace $T \in \mathcal{A}^*$ on global states is the function denoted by $\Delta : \overline{Q} \times \mathcal{A}^* \to \overline{Q}$.

The states $q \in Q_i \setminus \{\bot_i\}$ such that $\delta_i(q) \in \mathcal{V}$ are called the *final states* of process i. Let $F_i \subseteq Q_i$ denote the set of final states of process i. A trace $T \in \mathcal{A}^*$ is *one-shot* if it contains at most one outer invocation i_i^x and one outer response r_i^x for each process i. In particular, there is no restriction on the number of inner actions, since the objects in Obj are not assumed to be one-shot. A one-shot trace $T \in \mathcal{A}^*$

is *terminating* if after executing it, each process that participates in T ends in a final state. More formally, if we write $q = \Delta(q_{\text{init}}, T)$, for each i such that i_i^x occurs in T, we must have $q_i \in F_i$. Note that in a valid terminating trace, no action r_i^x occurs: when a process is in a final state, the process is ready to return its output value, but it did not return it yet.

We can now define the simplicial protocol $(\mathcal{I}, \mathcal{P}, \Psi)$ associated to P.

Definition 2.72 (Simplicial protocol associated to P)**.** The protocol complex $\mathcal{P} = (V_{\mathcal{P}}, S_{\mathcal{P}}, \chi_{\mathcal{P}}, \ell_{\mathcal{P}})$ is defined as follows. The set of views is $\mathsf{Views} = \bigcup_i F_i$. The vertices are of the form (i, q_i) where $i \in [n]$ is a process number and $q_i \in F_i$ is a final state of process i. Such a vertex is colored by i and labeled by q_i. Finally, for each one-shot trace T that is valid and terminating, we get a simplex $Y_T = \{(i, q_i) \mid i \text{ participates in } T\} \in S_{\mathcal{P}}$, where q_i is the i-th component of $\Delta(q_{\text{init}}, T)$.

We now define the carrier map $\Psi : \mathcal{I} \to 2^{\mathcal{P}}$. For $X \in \mathcal{I}$ an input simplex, we let $S = \chi_{\mathcal{I}}(X)$ be the set of participating processes, and $(v_i)_{i \in S}$ their input values. Then, $\Psi(X)$ consists of all simplexes Y_T where T is a valid terminating one-shot trace such that every invocation that occurs in T is of the form $\mathsf{i}_i^{v_i}$ for some $i \in S$.

It is straightforward to see that $\mathcal{P}$ is chromatic, and that Ψ is monotonic and chromatic. To check that $\mathcal{P}$ is pure of dimension $(n-1)$, let Y_T be a simplex of $\mathcal{P}$. We want to extend it to an $(n-1)$-dimensional simplex. To do so, first we append at the end of the trace T invocations i_i^x for each process i that does not participate in T. Since we assumed that the protocol P is wait-free, we can run each of these processes until it reaches a final state. Thus, we get a trace T' that is valid and terminating, and the corresponding simplex $Y_{T'}$ is of dimension $(n-1)$ and contains Y_T. Checking that Ψ is rigid is similar.

2.4.4 A generalized asynchronous computability theorem

The *asynchronous computability theorem* of Herlihy and Shavit [70] states that a task Θ has a wait-free protocol using read/write registers if and only if there exists a chromatic subdivision of the input complex, and a chromatic simplicial map from this subdivision to the output complex, that satisfies some conditions. That statement actually combines two claims: (a) a given protocol P using read/write registers solves the task Θ if and only if there exists a chromatic simplicial map from the protocol complex of P to the output complex, satisfying some conditions; and (b) to study task solvability using read/write registers, we can restrict to a particular class of protocols (iterated immediate snapshots) whose protocol complexes are subdivisions of the input complex.

In this section, we will prove a generalization of claim (a), which works not only for read/write registers, but for protocols which are allowed to use any combination of arbitrary objects, as defined in Section 2.3.1. We will not have a counterpart of claim (b), since this characterization is specific to the particular case of protocols using read/write registers.

Let $\Theta = (\mathcal{I}, \mathcal{O}, \Xi)$ be a task, $P = (P_i)_{i \in [n]}$ a protocol, and $(\mathcal{I}, \mathcal{P}, \Psi)$ its associated simplicial protocol as described in the previous section. Notice that, since we defined the vertices of $\mathcal{P}$ to be pairs (i, q_i) where $q_i \in F_i$ is a final state of process i, there is a map $\delta : V_{\mathcal{P}} \to \mathcal{V}$ defined as $\delta(i, q_i) = \delta_i(q_i)$, where δ_i is the decision function of the program P_i.

The following Theorem gives a topological characterization of what it means for the protocol P to implement (in the sense of Definition 2.46) the task Θ:

Theorem 2.73 (Generalized Asynchronous Computability Theorem)**.** *The protocol P implements the task-object $G(\Theta)$ if and only if the map $\delta : V_{\mathcal{P}} \to \mathcal{V}$ induces a chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Psi$ is carried by Ξ.*

Proof. In this proof, to distinguish between the execution traces of P (on the alphabet $\{\mathsf{i}_i^x, \mathsf{r}_i^x, \mathsf{i}(o)_i^x, \mathsf{r}(o)_i^x\}$) and the outer traces (on the alphabet $\{\mathsf{i}_i^x, \mathsf{r}_i^x\}$), we use lowercase letters t, t', s, s' for the former and capital letters T, T', S, S' for the latter.

($\Rightarrow$): Assume that P implements the object $G(\Theta)$, i.e., every trace $T \in \llbracket P \rrbracket$ satisfies the conditions of Definition 2.61. First, we want to define the map $\delta : V_{\mathcal{P}} \to V_{\mathcal{O}}$. Let $(i, q_i) \in V_{\mathcal{P}}$ be a vertex of $\mathcal{P}$; we would like to take $\delta(i, q_i) = (i, \delta_i(q_i))$, but we do not know yet that it is a vertex of $\mathcal{O}$. The vertex (i, q_i) belongs to a $(n-1)$-dimensional simplex Y_t for some execution trace t which is valid, one-shot and terminating. Let $q = \Delta(q_{\text{init}}, t)$ be the global state after executing t. In particular, the i-th component of q is q_i. For each $j \in [n]$, write $d_j = \delta_j(q_j)$ the value that process j is about to decide in the trace t. We also write $v_j \in \mathcal{V}$ the input value of process j in t. Let t' denote the trace obtained by appending responses $\mathsf{r}_j^{d_j}$ at the end of t. Then t' is still a valid trace, so if we write $T = \pi(t)$ its projection, we get $T \in \llbracket P \rrbracket \subseteq G(\Theta)$. Since T is a prefix of itself, and T is a complete trace, that means it respects the task specification, i.e., $\{(j, d_j) \mid j \in [n]\} \in \Xi(\{(j, v_j) \mid j \in [n]\})$. In particular, (i, d_i) is in the image of Ξ, so it is a vertex of $\mathcal{O}$.

The map $\delta : V_{\mathcal{P}} \to V_{\mathcal{O}}$ is chromatic since it sends i-colored vertices to i-colored vertices. Let us show that it is a simplicial map. Let $Y_t \in S_{\mathcal{P}}$ be a simplex of $\mathcal{P}$. Let $S \subseteq [n]$ be the set of processes participating in t, then Y_t is of the form $Y_t = \{(i, q_i) \mid i \in S\}$, and $\delta(Y_t) = \{(i, d_i) \mid i \in S\}$. As we did in the previous paragraph, we can add responses $\mathsf{r}_i^{d_i}$ for $i \in S$ at the end of the trace t, to obtain a complete valid trace whose projection is in $\llbracket P \rrbracket$, and thus also in $G(\Theta)$. This implies that $\delta(Y_t)$ is in the image of Ξ, so it is a simplex of $\mathcal{O}$.

Finally, we need to show that $\delta \circ \Psi$ is carried by Ξ. Let $X \in \mathcal{I}$ be an input simplex, and let $Z \in (\delta \circ \Psi)(X)$ be an output simplex. Then Z must be of the form $\delta(Y_t)$ for some $Y_t \in \Psi(X)$. We write $S \subseteq [n]$ the set of participating processes of X, and $(v_i)_{i \in S}$ their input values. So, t is a valid terminating one-shot trace such that every invocation in t is of the form $\mathsf{i}_i^{v_i}$ for some $i \in S$. Let $S' \subseteq S$ be the participating set of t. Let $q = \Delta(q_{\text{init}}, t)$ be the global state after executing t; we have $Y_t = \{(i, q_i) \mid i \in S'\}$, and $\delta(Y_t) = \{(i, d_i) \mid i \in S'\}$. Once again, we can append appropriate responses $\mathsf{r}_i^{d_i}$ to the trace t, and then we obtain $\pi(t') = T \in \llbracket P \rrbracket \subseteq G(\Theta)$. So, we obtain $\delta(Y_t) \in \Xi(\{(i, v_i) \mid i \in S'\})$. Since $\{(i, v_i) \mid i \in S'\} \subseteq X$, by monotonicity of Ξ, we finally get $\delta(Y_t) \in \Xi(X)$.

($\Leftarrow$): Assume that the induced map $\delta : V_{\mathcal{P}} \to V_{\mathcal{O}}$ defined as $\delta(i, q_i) = (i, \delta_i(q_i))$ is a chromatic simplicial map from $\mathcal{P}$ to $\mathcal{O}$, and that $\delta \circ \Psi$ is carried by Ξ. We want to prove that P implements the object $G(\Theta)$, i.e., that $\llbracket P \rrbracket \subseteq G(\Theta)$. Let $T \in \llbracket P \rrbracket$ be a one-shot outer trace; to show that $T \in G(\Theta)$, we take a prefix S of T. Since $\llbracket P \rrbracket$ is prefix-closed by Theorem 2.54 we also have $S \in \llbracket P \rrbracket$; i.e., there is an execution trace s which is valid for P and such that $\pi(s) = S$. Our first goal is to complete the trace S, that is, find valid responses to the pending invocations of S. Since the protocol P is wait-free, we can just run the pending processes one by one until they all reach a final state: formally, this amounts to appending inner actions to the trace s according to its program, until a final state q_i is reached. Then, we add the appropriate response $\mathsf{r}_i^{\delta_i(q_i)}$ to obtain a trace s', which extends s, is still valid and one-shot, and which does not have pending invocations. The projection $S' = \pi(s') \in \llbracket P \rrbracket$ is an extension of S where we added the responses $\mathsf{r}_i^{\delta_i(q_i)}$ to the pending invocations of S.

Now that we have S', we need to prove that $Z_{S'} \in \Xi(X_{S'})$, where $X_{S'}$ is the input simplex defined by $X_{S'} = \{(i, v_i) \mid \mathsf{i}_i^{v_i} \text{ occurs in } S'\}$, and $Z_{S'}$ is the output simplex $Z_{S'} = \{(i, d_i) \mid \mathsf{r}_i^{d_i} \text{ occurs in } S'\}$. Our goal is now to decompose $Z_{S'}$ as $Z_{S'} = \delta(Y)$ for some $Y \in \Psi(X_{S'})$. We write s'' for the trace obtained by removing from s' all the outer responses, i.e., actions of the form $\mathsf{r}_i^{d_i}$. We claim that s'' is still a valid trace. Indeed, no action from process i can occur after the outer response (otherwise, s' would not be one-shot), and the only effect of $\mathsf{r}_i^{d_i}$ is to change the local state of i, which does not affect the validity of the actions of other processes. Moreover, in the global state $q = \Delta(q_{\text{init}}, s'')$, every process that participates in s'' is in a final state, in other words, s'' is terminating. Thus, we have a simplex $Y_{s''} \in S_{\mathcal{P}}$ in the protocol complex, consisting of all the vertices (i, q_i) where i participates in s''. Since s'', s' and S' all have the same participating set, $Y_{s''} \in \Psi(X_{S'})$. And since $d_i = \delta_i(q_i)$ (because in s'', process i is ready to decide $\mathsf{r}_i^{d_i}$), we have $\delta(Y_{s''}) = Z_{S'}$. This decomposition of $Z_{S'}$ shows that $Z_{S'} \in \delta \circ \Psi(X_{S'})$. Since we assumed that $\delta \circ \Psi$ is carried by Ξ, this implies $Z_{S'} \in \Xi(X_{S'})$, which concludes the proof. $\square$

Despite the verbosity of the proof, nothing complicated is going on: we are just putting together all the definitions of the chapter. In particular, the crucial definitions that allow the proof to go through are the expansion property (5) in Definition 2.22; the map G (Definition 2.61) that characterizes the objects that correspond to tasks; and the definition of the protocol complex $\mathcal{P}$ associated to a protocol P in Section 2.4.3.

If we instantiate Theorem 2.73 with Obj containing only an iterated immediate snapshot object, combined with the fact that immediate snapshot protocol complexes are subdivisions of the input complex, we obtain Herlihy and Shavit's asynchronous computability theorem for read/write registers. If our set Obj of objects contains only linearizable objects, then we obtain Theorem 2.19, our first generalization of the asynchronous computability theorem that we obtained in the "first approach" Section 2.1. Thus, Theorem 2.73 subsumes both these weaker versions, since it can deal with arbitrary (not-necessarily linearizable) objects.

In general, if we fix a particular set of objects Obj, to prove that a task cannot be solved using the objects of Obj, one needs to find a topological invariant that holds in *any* protocol complex $\mathcal{P}$ associated to any protocol P. This is usually where all the difficulty of the proof lies, and of course Theorem 2.73 does not help with that part. What the Theorem says is merely that solvability in the sense of protocol complexes agrees with the more basic notion of solvability defined in Section 2.3.

Conclusion. In Sections 2.2, 2.3 and 2.4, we have extended Herlihy and Shavit's asynchronous computability theorem, which gives a topological characterization of the solvability of tasks by wait-free protocols, not only in the context of read/write registers, but for a large class of arbitrary objects. In the usual topological approach to fault-tolerant computability, as described for example in [64], the existence of a simplicial map from the protocol complex to the output complex is taken as the *definition* of task solvability. Thanks to Theorem 2.73, we now know that this definition is *sound*, in the sense that it actually reflects our concrete understanding of what it means to solve a task. In particular, our construction of the protocol complex in Section 2.4.3 is a valuable tool to define protocol complexes in a systematic way. Given a particular object of interest, instead of trying to think about what the right notion of "view" should be, we can instantiate Definition 2.72 to obtain a protocol complex, which we proved to be the correct one.

There are still several ways in which we could generalize this theorem. Firstly, it would be nice to relax the “wait-free” assumption, in order to obtain a *t-resilient* Asynchronous Computability Theorem for arbitrary objects, extending the work of [113] which is restricted to read/write protocols. We expect that this should not be too difficult to do: it would mainly require some technical overhead to adapt all of our definitions, but no conceptual difficulty should arise. A more challenging extension would be to go beyond tasks, for example using the notion of *refined task* [18], which can express all one-shot objects, or even *long-lived tasks* [22], which can be invoked multiple times. Doing so would probably require a truly novel idea, since the usual notion of protocol complex is by definition “frozen in time”.

In the next section, we explore more in depth one property of our trace semantics: *compositionality*. In computer science, complex programs are usually built in a modular way: smaller components are combined to obtain more complex ones, and so on. Our point of view of “objects implementing objects” in Definition 2.46, instead of the traditional “objects solving tasks” point of view, opens the door to such a compositional approach.

2.5 Towards game semantics for fault-tolerant protocols

Remark 2.74. This section aims at presenting the framework of game semantics to people from the field of distributed computability, who might not be aware of it. Thus, we do not assume any prior knowledge of game semantics from the reader, and very limited knowledge of functional programming. As a consequence, we do not dig very deep into the formal constructions, and instead explain things at an intuitive level. The technical part of the section will consist mostly of definitions, reformulating the computational model of Section 2.3 into the language of games and strategies. Finding actual applications of this point of view to distributed computability is still work in progress.

In this section, we study the compositionality properties of the computational model that we presented in Section 2.3, using the formalism of *game semantics*. To define the semantics of a protocol in Definition 2.46, we first considered execution traces with *inner actions* and *outer actions*, and then, in order to “forget” how the protocol was implemented and see it as a black box, we removed all the inner actions. This operation is quite reminiscent of what happens when we compose strategies in game semantics: to compose a strategy $\sigma : A \to B$ with a strategy $\tau : B \to C$, we “forget” about what happens in the B component and obtain a strategy $\tau \circ \sigma : A \to C$.

2.5.1 What is game semantics?

Originally, game semantics [75, 2] came up in the setting of sequential programs, and more precisely, functional programming languages. These languages are characterized by their type system and the use of higher-order functions, that is, functions that can take other functions as argument. A basic language to study such programs is the simply-typed λ-calculus, or extensions of it such as PCF.

In PCF, there are three basic types of data: Nat, the type of *natural numbers*, which contains values $0, 1, 2, \ldots$; Bool, the type of *booleans*, which contains the two values true and false; and Unit, the *unit* type, which contains only one value $*$. Moreover, given two types T and U, we can create the *product type* $T \times U$, which contains pairs of elements of T and U, and the *function type* $T \to U$, which contains functions from T to U. To manipulate this data, we are allowed to use basic programming constructs, namely, “if - then - else” statements and recursion on natural numbers.

For instance, we can form the type $\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$; a program of this type takes two arguments as input, each of them a boolean, and returns a boolean. For example, the following two programs `p` and `q` are of type $\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$. They both implement the "or" function on booleans.

```
let p (b1,b2) =
  if (b1 == true)
  then return true
  else if (b2 == true)
       then return true
       else return false
```

```
let q (b1,b2) =
  if (b1 == true)
  then if (b2 == true)
       then return true
       else return true
  else if (b2 == true)
       then return true
       else return false
```

A very simple semantics for PCF is the set-theoretic one: the types Nat, Bool and Unit are interpreted as sets, $\mathbb{N}$, $\{\mathsf{true}, \mathsf{false}\}$ and $\{*\}$, respectively. The type constructor $T \times U$ is interpreted as the cartesian product, and the type $T \to U$ as the set of functions. In this set-theoretic model, the two programs `p` and `q` are given the same interpretation: they correspond to the set-theoretic "or" function $f(b_1, b_2) = b_1 \vee b_2$. It is well known to all computer scientists that programs contain much more information than set-theoretic functions: in computer science, notions such as algorithmic complexity or computational effects come into play. Different kinds of semantics can put the emphasis on different aspects of programming languages. In the case of game semantics, the notion that it aims to capture is called *observational equivalence*: we view the two programs `p` and `q` above as black boxes, consider some client code that is allowed to call these programs. Is it able to observe the difference between `p` and `q` from the outside?

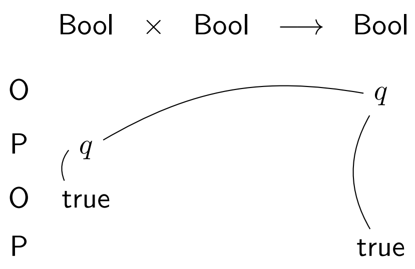
The answer is yes, as long as we are considering *call-by-name* computation, as opposed to *call-by-value*. In a call-by-name programming language, the arguments given to a function (such as b_1 and b_2) are not computed before the function is called, but later, when the function is actually using its argument. Thus, we can take the following program `t` of type Bool,

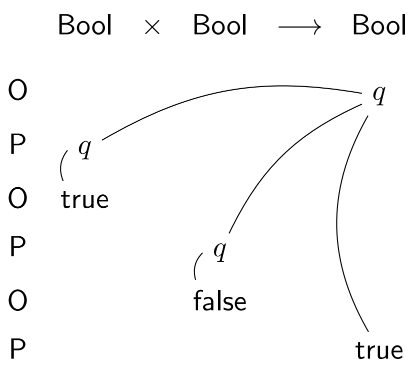
```
let t =
  print("Hello World!");
  return true
```

and use it to exhibit different behaviors for `p` and `q`. Indeed, the computation `p(true,t)` will not print the message, since the second argument is never evaluated, but `q(true,t)` will evaluate both arguments and print the message.

The central concept of game semantics is the notion of interaction between a *program* and its *environment*. In the above example, the "environment" simply consists of the arguments that are given to the program: are they accessed by the program? How many times? In what order? To be able to make such distinctions, the programs are not interpreted as set-theoretic functions, but as sets of *execution traces*. Moreover, these execution traces are viewed as *plays* in a two-player game, where the program, called P (for "Player"), plays against its environment, called O (for "Opponent"). The programs will correspond to *strategies* in this game, that is, sets of plays satisfying some properties.

In the picture below are depicted the plays that occur when we compute `p(true,false)` (depicted on the left) and `q(true,false)` (depicted on the right). They are read from top to bottom, and can be understood as follows. In the play on the left, the opponent starts by asking a *question* (written as q) to the program: “what value do you return?”. Then, the player answers that question with another question: “what is my first argument?”. The opponent answers “true” to this question, and thus the player can also answer “true” to the initial question. The depicted edges are called *pointers*; when one of the two players plays a move, it is directly responding to one of the previous moves from the other player, which is not necessarily the last one. The pointers simply keep track of which moves respond to each other. In the play on the right, the first three moves are the same, but then the player does not give his answer right away, and asks another question: “what is my second argument?”. The opponent answers “false”, and the player finally answers the initial question by “true”.





A play, as represented in the pictures above, represent single executions of the program. The program itself is interpreted as a *strategy* for player P, that is, a set of plays. Intuitively, this set specifies, for every possible sequence of moves from the opponent, what the player’s next move will be.

2.5.2 Back to our computational model

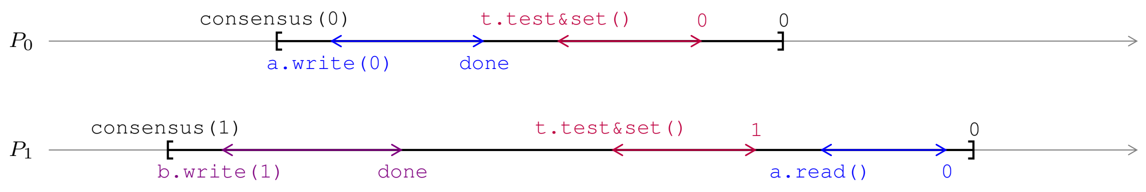
We now explain informally how game semantics relates to the computational model of Section 2.3. First, notice that the previous section deals with sequential programs only. There exist many approaches that aim at extending game semantics to concurrent programs [92, 93, 23]. These works usually consider a very general setting, where an unbounded number of processes can be created and terminated at any time during the computation. In our case, we are interested in a much simpler kind of concurrent programs: a fixed number n of processes are run in parallel. The main thing we have to do is, given n sequential strategies corresponding to the programs run by the processes, to find a way to make them interact in parallel. So, we can hope to avoid many of the technical difficulties that occur in concurrent game semantics, and instead find a simple adaptation of the usual sequential setting.

We will illustrate our idea using the example of Section 2.3.1, where two processes solve consensus using two shared registers and one test-and-set object.

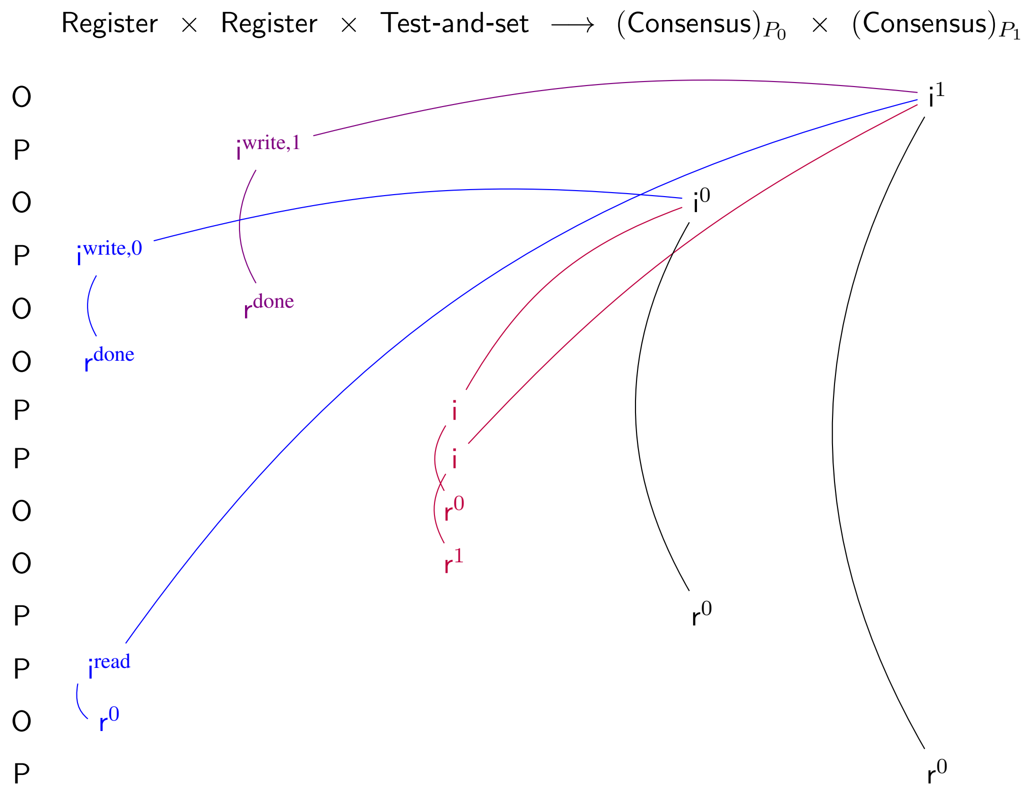
Object types. The *type* of an object will simply be its signature, that is, the types of the methods of the object. For example, a read/write register has two methods, `write` : $\mathsf{Nat} \to \mathsf{Unit}$ (which takes a natural number as argument and does not return a value), and `read` : $\mathsf{Unit} \to \mathsf{Nat}$ (which takes no argument and returns a natural number). The corresponding type is $\mathsf{Register} := (\mathsf{Nat} \to \mathsf{Unit}) \times (\mathsf{Unit} \to \mathsf{Nat})$. Similarly, the type of the test-and-set object is $\mathsf{Test\text{-}and\text{-}set} := \mathsf{Unit} \to \mathsf{Bool}$, since it takes no argument and returns a value 0 or 1, which we identify with booleans. The type of consensus is $\mathsf{Consensus} := \mathsf{Nat} \to \mathsf{Nat}$, it takes as argument the value that the process proposes, and returns the agreed-upon value.

Concurrent strategies. The example program of Section 2.3.1 is an implementation of a consensus object. Moreover, this program is making use of two registers and one test-and-set object, which have been implemented previously. This program should still work given any particular implementation of these three objects. So, we can see it as a program of type Register $\times$ Register $\times$ Test-and-set $\to$ Consensus, meaning that it is parameterized by two registers and one test-and-set object, and it implements a consensus object. This is the type of the program that is run by one single process, and we could easily interpret it as a sequential strategy using the standard sequential framework of game semantics.

Now, our goal is to define the executions of the *protocol*, that is, the two programs running together and interacting by calling the shared objects. Remember how we defined in Section 2.3.2 the set $\mathcal{T}_P$ of *valid executions* of the protocol, using so-called *inner actions* and *outer actions*. A valid execution is represented of our consensus protocol in the picture below.



Recall that, formally, the outer invocations and responses (in black, square brackets) are of the form i_i^x and r_i^x, while the inner invocations and responses (colored, angle brackets) are of the form $\mathsf{i}_i^x(o)$ and $\mathsf{r}_i^x(o)$, where i is a process number, x is an input or output value, and o is an object. So, the trace above can be written $\mathsf{i}_1^1 \cdot \mathsf{i}_1^{\text{write},1}(b) \cdot \mathsf{i}_0^0 \cdot \mathsf{i}_0^{\text{write},0}(a) \cdot \mathsf{r}_1^{\text{done}}(b) \cdots$, and so on. If we rotate the picture by $90°$, we can view it as the following *concurrent play*, where time flows from top to bottom:



Register × Register × Test-and-set ⟶ (Consensus)$_{P_0}$ × (Consensus)$_{P_1}$
O
P
O
P
O
O
P
P
O
O
P
P
O
P
i^1
$\mathsf{i}^{\text{write},1}$
i^0
$\mathsf{i}^{\text{write},0}$
r^{done}
r^{done}
i
i
r^0
r^1
r^0
i^{read}
r^0
r^0

This is still a two-player game: both processes have the role of the player, with label P, while the opponent represents the environment, with label O. The type Consensus on the right has been duplicated in two components, corresponding to the two processes P_0 and P_1. Notice how we have removed some informations from the invocation and response symbols. More precisely, the process number i and the object o have been omitted in the symbols i_i^x, r_i^x, $\mathsf{i}_i^x(o)$ and $\mathsf{r}_i^x(o)$. Indeed, they can be recovered as follows. The object o of an inner action is obtained by looking vertically in which of the components it occurs. The process number i of an action is recovered by following the pointers upwards in order to find in which component, $(\mathsf{Consensus})_{P_0}$ or $(\mathsf{Consensus})_{P_1}$, the thread started.

Another thing we can remark about this play, is that what is usually called "questions" and "answers" in game semantics corresponds to "invocations" and "responses" in our execution traces. The O/P polarity of the moves is a bit more subtle. The player moves are the ones that correspond to actions that the programs *choose* to do: invoke one of the shared objects (inner invocation), or decide on an output value (outer response). The opponent moves are the ones coming from the environment: the outer invocations, which give its argument to the program, and the inner responses, which depend on the behavior of the shared objects. Notice how we no longer have an alternation between opponent and player moves. However, we can recover this alternation if we consider only the moves coming from one of the components $(\mathsf{Consensus})_{P_i}$. In fact, we can decompose this concurrent play into two sequential plays, one containing the moves of process 0, and the other containing the moves of process 1. These sequential plays belong to the sequential strategies of the programs.

Lastly, by looking vertically at the first column (corresponding to register $\mathtt{a}$), we can see how the two processes communicate by using the shared register: first one of them writes 0, then the other reads and obtains value 0. However, in a *concurrent strategy*, the player should consider every possible move from the opponent. So, a strategy will also contain plays where the register does not behave as expected. This is contrasted with what we did in Definition 2.44, where the specification of the objects was given in advance. Here, if we want to take into account the fact that we actually have registers and test-and-set objects, we need to *precompose* with an implementation of these objects. For instance, if we have three protocols of type $A \to \mathsf{Register}$, $B \to \mathsf{Register}$, and $C \to \mathsf{Test\text{-}and\text{-}set}$, implementing each shared object using other objects A, B, C respectively, we obtain after composition a protocol of type $A \times B \times C \to \mathsf{Consensus}$, implementing consensus from the three objects A, B and C.

2.5.3 A game semantics for fault-tolerant protocols

We now formalize the game model that we sketched in the previous section.

A simple programming language

First, we define formally the syntax of a toy programming language, which is powerful enough to express the examples that we are interested in.

Types.

$$\tau \quad ::= \quad \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{Unit} \mid \tau \times \tau \mid \tau \to \tau$$

Arithmetic expressions. We now fix a set Var of *local variables*, written as $x, y, z \in \mathsf{Var}$. A variable denotes a value of basic type (Unit, Bool or Nat), or a tuple of such values, which is held locally by a process.

$$\begin{array}{rcll} e & ::= & x \mid * \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid 2 \mid \dots & x \in \mathsf{Var} \\ & & \mid e \vee e \mid e \wedge e \mid e + e \mid e \times e \mid \dots & \\ & & \mid \langle e, e \rangle \mid \mathsf{proj}_1(e) \mid \mathsf{proj}_2(e) & \end{array}$$

An *arithmetic expression* is just a local computation made by some process. It can use local variables as well as constants, and combine them using the usual operations on booleans and natural numbers.

Programs. We also fix and a set Obj of *objects names*, written $a, b, c \in \mathsf{Obj}$. They denote shared objects, whose type is usually a function type or product of functions. When it is a product of functions, each component is called a *method*. A *program* run by one process implements all the methods of the object that is being implemented:

$$\begin{array}{rcll} P & ::= & \mathsf{method}_1(v)\ \{\ C\ \} & v \in \mathsf{Var} \\ & & \mathsf{method}_2(v)\ \{\ C\ \} & \\ & & \dots & \\ & & \mathsf{method}_k(v)\ \{\ C\ \} & \end{array}$$

The body C of each method implementation is given by the following grammar:

$$\begin{array}{rcll} C & ::= & \mathsf{return}(e) & \\ & & \mid x := a.\mathsf{call}_i(e);\ C & x \in \mathsf{Var},\ a \in \mathsf{Obj},\ i \in \mathbb{N} \\ & & \mid \mathsf{if}\ e\ \mathsf{then}\ C\ \mathsf{else}\ C & \end{array}$$

So, a program can either:

– return a value using $\mathsf{return}(e)$, where e is an arithmetic expression, or

– call the i-th method of object a, with input value e, and store the result in variable x, or

– branch depending on its local state to choose the next call.

The arithmetic expressions e are allowed to use the variables that have been previously defined, as well as the input value v of the method. Note that for simplicity we do not include while loops; this is of course restrictive, but not necessary for the examples that we are interested in. In particular, if the object calls always terminate, every such program is wait-free. Another simplifying assumption is that we cannot nest object calls: we cannot write something like `a.write(t.test-and-set())`. Instead, we have to separate the two object calls as follows, `x := t.test-and-set(); a.write(x)`. This forces a call-by-value computation of object calls, and avoids dealing with issues like the evaluation order of arithmetic operations.

Protocol. Finally, a protocol $\mathcal{P}$ for a fixed number n of processes declares the types of the shared objects $a_1, \dots, a_k$, as well as the type of the object that is being implemented, and it gives one program P_i for each of the n processes.

$$\begin{aligned}\mathcal{P} \quad ::= \quad & \mathsf{shared}\ a_1 : \tau_1 \\ & \dots \\ & \mathsf{shared}\ a_k : \tau_k \\ & \mathsf{implements}\ \tau \\ & P_1 \,\|\, P_2 \,\|\, \dots \,\|\, P_n\end{aligned}$$

Typing rules. Such a protocol $\mathcal{P}$ is *well-typed* if, in the context $\Gamma = a_1 : \tau_1, \dots, a_k : \tau_k$, the program P_i of each process is of type τ. The typing rules for programs are the following:

$$\textsc{Ret}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : \tau} \qquad \textsc{Call}\ \frac{\Gamma \vdash a : \Pi_i \sigma_i \to \tau_i \qquad \Gamma \vdash e : \sigma_i \qquad \Gamma, x : \tau_i \vdash C : \tau}{\Gamma \vdash x := a.\mathsf{call}_i(e);\ C : \tau}$$

$$\textsc{If}\ \frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash C_1 : \tau \qquad \Gamma \vdash C_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \tau} \qquad \textsc{Obj}\ \frac{}{\Gamma, a : \tau \vdash a : \tau} \qquad \textsc{Prog}\ \frac{\Gamma, v : \sigma_i \vdash C : \tau_i}{\Gamma \vdash \mathsf{method}_i(v)\ \{\ C\ \} : \sigma_i \to \tau_i}$$

The typing rules for arithmetic expressions are defined as expected.

Example 2.75. The consensus implementation of Section 2.3.1 for two processes is given by:

$$\begin{aligned}& \mathsf{shared}\ a : (\mathsf{Unit} \to \mathsf{Nat}) \times (\mathsf{Nat} \to \mathsf{Unit}) \\ & \mathsf{shared}\ b : (\mathsf{Unit} \to \mathsf{Nat}) \times (\mathsf{Nat} \to \mathsf{Unit}) \\ & \mathsf{shared}\ t : \mathsf{Unit} \to \mathsf{Bool} \\ & \mathsf{implements}\ \mathsf{Nat} \to \mathsf{Nat}\end{aligned}$$

```
P0: consensus(v) {
      x := a.write(v);
      y := t.test-and-set(*);
      if y then
        return v;
      else
        z := b.read(*);
        return z;
    }
```

```
P1: consensus(v) {
      x := b.write(v);
      y := t.test-and-set(*);
      if y then
        return v;
      else
        z := a.read(*);
        return z;
    }
```

For readability, instead of numbering the methods, we gave them usual names: the two methods of the objects a and b are “read” and ”write”, the only method of the object t is “test-and-set”, and the only method of the object being implemented (let us call it c) is “consensus”. Notice however that nothing indicates that the objects a, b, t behave like read/write registers or test-and-set objects. The only restriction is about their types; but of course, this is far from enough to ensure that this protocol will indeed solve consensus. If we write τ_a the type of object a, the type of this protocol is $\tau_a \times \tau_b \times \tau_t \to \tau_c$, meaning that it is an implementation of an object of type $\tau_c = \mathsf{Nat} \to \mathsf{Nat}$, which is parameterized by three shared objects of type τ_a, τ_b, τ_t. The behavior of this protocol depends on a particular implementation of the three objects that we plug into it.

Remark 2.76. This syntax is very unusual in game semantics literature, which usually prefers λ-calculus notations. However, except for the parallel operator $P_1 \parallel P_2 \parallel \ldots \parallel P_n$, everything else is just syntactic sugar for usual functional programming constructions. A protocol $\mathcal{P}$ simply introduces object names with their types, i.e., it is a sequence of lambda-abstractions $\lambda a_1 \ldots \lambda a_n.\ P_i$. In programs, the $\mathsf{return}(e)$ operation would simply be written e in λ-calculus, and the $x := a.\mathsf{call}_i(e); C$ operation corresponds to the "let expression" $\mathsf{let}\ x = (\mathsf{proj}_i\ a)\ e\ \mathsf{in}\ C$. From the point of view of game semantics, the main interest of what we do in this section is the interpretation of the parallel operator.

Notice that we do not use the full power of higher-order functions: the type of programs has order 1 (functions between basic types), and the type of protocols has order 2 (functions between functions). We now define a *game semantics* for these protocols, keeping in mind that we want to understand the *composition* of protocols. Given a protocol of type $\tau \to \tau'$ and a protocol of type $\tau' \to \tau''$, their composition has type $\tau \to \tau''$. Can we define its behavior in terms of the behaviors of its two components?

Sequential game semantics

First, we recall the usual definitions of *arenas* and *strategies* from game semantics. We use them to interpret programs, that is, the sequential piece of code that is run by a single process. All the notions defined in this section are standard, see e.g. [75] or [2] for more details.

Arenas. The games of game semantics are played on *arenas*. An arena defines the set of possible moves for both players; they will be used to interpret the types of our programming language.

Definition 2.77 (Arena)**.** An *arena* is a labeled oriented forest $A = (M, \vdash, \lambda^{\mathsf{OP}}, \lambda^{\mathsf{QA}})$, where $\lambda^{\mathsf{OP}} : M \to \{\mathsf{O}, \mathsf{P}\}$ and $\lambda^{\mathsf{QA}} : M \to \{\mathsf{Q}, \mathsf{A}\}$ such that:
- If m is a root, then $\lambda^{\mathsf{OP}}(m) = \mathsf{O}$ and $\lambda^{\mathsf{QA}}(m) = \mathsf{Q}$,
- If $m \vdash m'$, then $\lambda^{\mathsf{OP}}(m) \neq \lambda^{\mathsf{OP}}(m')$,
- If $m \vdash m'$ and $\lambda^{\mathsf{QA}}(m') = \mathsf{A}$, then $\lambda^{\mathsf{QA}}(m) = \mathsf{Q}$.

Elements of M are called *moves*, the roots are *initial moves*. When $m \vdash m'$, we say that m *justifies* m'. If $\lambda^{\mathsf{OP}}(m) = \mathsf{P}$ (resp., O), we say that m is a *Player move* (resp., *Opponent move*). If $\lambda^{\mathsf{QA}}(m) = \mathsf{Q}$ (resp., A), we say that m is a *Question* (resp., an *Answer*).

Example 2.78. We can define arenas corresponding to the basic types:
- The empty arena is written $\mathbf{1}$.
- The singleton arena with one OQ-labeled move is written $\mathbf{0}$.
- The arena $\mathbf{B}$ of booleans has the set of moves $\{q, \mathsf{true}, \mathsf{false}\}$ where q is an Opponent Question, and true and false are Player Answers such that $q \vdash \mathsf{true}$ and $q \vdash \mathsf{false}$.
- The unit arena $\mathbf{U}$ has moves $\{q, *\}$, where q is an Opponent Question, and $*$ is a Player Answer with $q \vdash *$.
- The arena $\mathbf{N}$ of natural numbers has moves $\{q\} \cup \mathbb{N}$ where q is an Opponent Question, and every $n \in \mathbb{N}$ is a Player Answer with $q \vdash n$.

Definition 2.79 (Product arena)**.** The product $A \times B$ of two arenas is given by their disjoint union.

Definition 2.80 (Arrow arena)**.** Given two arenas A and B, the arena $A \Rightarrow B$ is obtained by taking their disjoint union; adding the edges $m \vdash m'$ where m and m' are initial moves of B and A respectively; and reversing the OP-polarity of A.

The above operations are usually sufficient to interpret λ-calculus. However, in our case, we also need a slightly different version of the arrow, for types of order 1. Indeed, we care about how many times and in what order a protocol is calling the shared objects; but we do not care about how many times it uses its input value. So we define a call-by-value version of the arrow arenas for first order types. This definition is inspired from [74], but we do not use their categorical constructions here; this can simply be seen as an unusual interpretation of our base types, the category constructed above it is the usual call-by-name one.

Example 2.81 (Cbv arrow arena). The arena $\mathbf{N} \overset{\text{cbv}}{\Longrightarrow} \mathbf{N}$ has the set of moves $\{0, 1\} \times \mathbb{N}$ (i.e., two disjoint copies of $\mathbb{N}$), where the moves of the form $(0, n)$ are Opponent Questions, and the moves of the form $(1, n)$ are Player Answers. Moreover, for every $n, m \in \mathbb{N}$, $(0, n) \vdash (1, m)$.
We define similar arenas for all first order function types, for instance the arena $\mathbf{B} \times \mathbf{N} \overset{\text{cbv}}{\Longrightarrow} \mathbf{U}$ has Opponent Questions of the form $(0, b, n)$ for $b \in \{\mathsf{true}, \mathsf{false}\}$ and $n \in \mathbb{N}$ and one Player Answer, $(1, *)$.

Strategies. A *strategy* for player P on a given arena is a set of plays, which specifies what is the player's response to the possible moves played by the Opponent.

Definition 2.82 (Pointed sequence)**.** A *pointed sequence* on an arena A is a pair (s, f) where $s = s_1 \cdots s_k$ is a sequence of moves in A, and $f : \{1, \dots, k\} \to \{0, \dots, k-1\}$ is a function such that:

- $f(i) < i$,
- if $f(i) = 0$, then s_i is an initial move,
- if $f(i) = j \neq 0$, then $s_j \vdash s_i$.

When $f(i) = j$, we say that s_i *points* to s_j or that s_j *justifies* s_i in (s, f). When $f^p(i) = j$ for some $p > 0$, we say that s_i points *hereditatily* to s_j (or s_j justifies *hereditarily* s_i) in (s, f). There is an abuse of notation here, when we write s_i to refer to the indexed occurrence (i, s_i) of the move s_i at position i in s.

When manipulating these sequences, we often omit the pointers and just write s for the pointed sequence (s, f). For example, if we write $s = s'$, it is implied that both the sequences and the pointers are equal.

Definition 2.83 (Play)**.** A *play* on an arena A is a pointed sequence s on A that is *alternating*: for all i, $\lambda^{\mathsf{OP}}(s_i) \neq \lambda^{\mathsf{OP}}(s_{i+1})$. The set of plays is written $\mathcal{P}_A$.

Given a pointed sequence s on $A \times B$ (or $A \Rightarrow B$), the *restriction* of s to A (resp., to B) is the sequence $s\restriction_A$ (resp., $s\restriction_B$) obtained by keeping only the moves from A (resp., from B). Note that in the case of $A \Rightarrow B$, an initial move of A points to a move of B in s, in which case, it loses its pointer in $s\restriction_A$. Notice that $s\restriction_A$ (resp., $s\restriction_B$) is a pointed sequence on A (resp., on B), but it is not necessarily a play.

We write $s' \sqsubseteq s$ when s' is a *prefix* of s (pointers included). We write $s' \sqsubseteq^{\mathsf{P}} s$ (resp., $s' \sqsubseteq^{\mathsf{O}} s$) to indicate that in addition s' ends with a Player (resp., Opponent) move. A play that ends with a Player move is called a P-play, and when $s' \sqsubseteq^{\mathsf{P}} s$, s' is a P-prefix of s (similarly for Opponent).

Definition 2.84 (Strategy)**.** A *strategy* σ on an arena A, written $\sigma : A$, is a set of P-plays on A that is closed under P-prefix: if $s' \sqsubseteq^{\mathsf{P}} s$ and $s \in \sigma$ then $s' \in \sigma$.

Definition 2.85 (Identity)**.** We define the strategy $\mathsf{id}_A : A \Rightarrow A$ as follows:

$$\mathsf{id}_A = \{s \in \mathcal{P}_{A_1 \Rightarrow A_2} \mid s \text{ is a P-play and for all } s' \sqsubseteq^{\mathsf{P}} s,\ \ s'\restriction_{A_1} = s'\restriction_{A_2}\}$$

where the indexes A_1, A_2 are only here to distinguish the two occurrences of A. It is easily checked that id_A is a strategy on $A \Rightarrow A$.

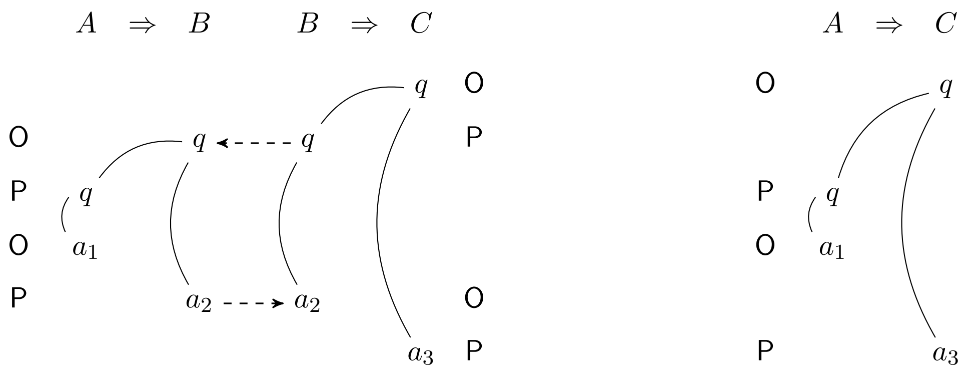
Definition 2.86 (Interaction)**.** Let A, B, C be arenas. An *interaction* u on A, B, C, written $u \in \mathrm{Int}(A, B, C)$, is a pointed sequence on the arena $(A \Rightarrow B) \Rightarrow C$ such that $u\restriction_{A,B}$, $u\restriction_{B,C}$ and $u\restriction_{A,C}$ are plays on $A \Rightarrow B$, $B \Rightarrow C$ and $A \Rightarrow C$, respectively. The three restrictions have the following meaning:

– $u\restriction_{A,B} = u\restriction_{A \Rightarrow B}$ as in the usual definition of restriction.
– $u\restriction_{B,C}$ is obtained by keeping the moves from B and from C.
– $u\restriction_{A,C}$ is obtained by keeping the moves from A and from C, and adding the following pointers: whenever, in u, a move a of A points to a move b of B; and b points to a move c of C, then a points to c in $u\restriction_{A,C}$. Note that in such a case, a, b, c are initial moves of A, B, C, which makes $u\restriction_{A,C}$ a pointed sequence on $A \Rightarrow C$.

Definition 2.87 (Composition)**.** Let $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ be two strategies. We define their *composition* $\sigma; \tau : A \Rightarrow C$ as follows (note that we write composition in the opposite order compared to the usual set-theoretic notation, $\sigma; \tau = \tau \circ \sigma$).

$$\sigma; \tau = \{u\restriction_{A,C} \mid u \in \mathrm{Int}(A, B, C) \text{ and } u\restriction_{A,B} \in \sigma \text{ and } u\restriction_{B,C} \in \tau\}$$

Composition of strategies can be depicted as follows, where the "B" component of the interaction u has been duplicated to see more clearly the two plays on $A \Rightarrow B$ and $B \Rightarrow C$.



Checking that $\sigma; \tau$ is indeed a strategy on $A \Rightarrow C$ requires some technical machinery of game semantics (the so-called *Zipping lemma*), that we do not detail here. Moreover, given strategies $\sigma : A \Rightarrow B$, $\tau : B \Rightarrow C$ and $\rho : C \Rightarrow D$, one can prove that $(\sigma; \tau); \rho = \sigma; (\tau; \rho)$, and that $\mathsf{id}_A; \sigma = \sigma$ and $\sigma; \mathsf{id}_B = \sigma$. Thus, arenas and strategies form a category. We write $\mathbf{Str}$ for the category whose objects are arenas and morphisms from A to B are strategies on $A \Rightarrow B$. The composition and identities are those defined above. The product of arenas $A \times B$ it is *not* a cartesian product in $\mathbf{Str}$, but it can be extended to a bifunctor $\times : \mathbf{Str} \times \mathbf{Str} \to \mathbf{Str}$. The empty arena $\mathbf{1}$ is the terminal object of $\mathbf{Str}$, and the neutral element of the product operation $\times$. The arrow $A \Rightarrow B$ can also be extended to a bifunctor $\Rightarrow : \mathbf{Str}^{\mathrm{op}} \times \mathbf{Str} \to \mathbf{Str}$, and for every arena A, the functor $(A \Rightarrow -)$ is right-adjoint to $(- \times A)$. In short:

Theorem 2.88 ([75])**.** *The category* $\mathbf{Str}$ *is a symmetric closed monoidal category.*

Remark 2.89. This structure makes it a suitable model to interpret λ-calculus (i.e., sequential programs). However, not all strategies correspond to the interpretation of some program! One can characterize exactly the set of strategies that correspond to some program: is a strategy is *deterministic*, *total*, *well-bracketed* and *innocent*, then it is the interpretation of some λ-term. In fact, we can do even better than this completeness result, which is called *full abstraction*: two λ-terms correspond to the same strategy if and only if they are *observationally equivalent*. We refer the reader to [75] for the definitions of all those notions, and the proofs of those theorems.

Concurrent game semantics

We know how to interpret sequential programs as strategies. To interpret protocols, we need a notion of *concurrent strategy*, which will correspond to n sequential strategies "running in parallel". Let $n \in \mathbb{N}$ be some fixed number of *processes*. An integer $p \in [n]$ is called a *process number*. Given an arena A, we write $TA := A^n$ for the n-ary product of A with itself. When we want to distinguish the components of TA, we write them $TA = A_0 \times \cdots \times A_{n-1}$. A *concurrent strategy* will be defined on arenas of the form $A \Rightarrow TB$, whose intended meaning is to represent n processes executing a function of type $A \Rightarrow B$ concurrently, where A is the type of some shared object.

Definition 2.90 (p-Restriction)**.** Let s be a pointed sequence on $A \Rightarrow TB$ and $p \in [n]$ be a process number. The *restriction* of s to process p, written $s{\upharpoonright_p}$, is obtained by keeping only the moves that are hereditarily justified by a move of B_p. Notice that $s{\upharpoonright_p}$ is a pointed sequence on $A \Rightarrow B$.

Definition 2.91 (Concurrent play)**.** A *concurrent play* on $A \Rightarrow TB$ is a pointed sequence s such that for all $p \in [n]$, $s{\upharpoonright_p}$ is a play.

Recall from Definition 2.83 that a play is simply a pointed sequence that is alternating, i.e., the moves alternate between Opponent and Player. Since we remarked that $s{\upharpoonright_p}$ is already a pointed sequence, Definition 2.91 amounts to requiring that each $s{\upharpoonright_p}$ be alternating. Note that a concurrent play is not in general a play, since it might not be globally alternating.

Definition 2.92 (Concurrent strategy)**.** A *concurrent strategy* on $A \Rightarrow TB$ is a set σ of concurrent P-plays which is closed under P-prefix and such that $\sigma{\upharpoonright_p}$ is a strategy on $A \Rightarrow B$ for all $p \in [n]$, where $\sigma{\upharpoonright_p} = \{s{\upharpoonright_p} \mid s \in \sigma$ and the last move of s is hereditarily justified by a move of $B_p\}$.

Definition 2.93 (Return)**.** Let A be an arena. The concurrent strategy $\eta_A : A \Rightarrow TA$ is defined as follows:

$$\eta_A = \{s \mid \forall p \in [n],\ s{\upharpoonright_p} \in \mathsf{id}_A\}$$

Proposition 2.94. *η_A is a concurrent strategy.*

Proof. Every pointed sequence $s \in \eta_A$ is a concurrent play, since each $s{\upharpoonright_p} \in \mathsf{id}_A$ is a play. Moreover, η_A is closed under P-prefix since id_A is closed under P-prefix. Finally, one can check that $\eta_A{\upharpoonright_p} = \mathsf{id}_A$ for all p, so it is indeed a strategy on $A \Rightarrow A$. $\square$

Definition 2.95 (Bind)**.** Let $\sigma : A \Rightarrow TB$ be a concurrent strategy. We define the concurrent strategy $\sigma^* : TA \Rightarrow TB$ as

$$\sigma^* = \{s^* \mid s \in \sigma\}$$

where s^* is defined by induction on the length of s as follows. If s is a pointed sequence on $A \Rightarrow TB$, s^* will be a pointed sequence on $TA \Rightarrow TB$.

– if $s = \varepsilon$ then $s^* = \varepsilon$,
– if $s = s'm$ where m is a move of B_p, then $s^* = (s')^*m$,
– if $s = s'm$ where m is a move of A that is hereditarily justified by a move in B_p, then $s^* = (s')^*m$, where m is played in A_p.

Proposition 2.96. *σ^* is a concurrent strategy.*

Proof. It can be checked by an easy induction that if s is a concurrent play on $A \Rightarrow TB$, then s^* is a concurrent play on $TA \Rightarrow TB$. Another induction shows that a prefix of s^* is of the form $(s')^*$, where s' is a prefix of s. Thus, since σ is closed under P-prefix, so is σ^*. Finally, in $\sigma^* \upharpoonright_p$, which is a set of concurrent plays on $TA \Rightarrow B$, every move played in TA is actually played in A_p, by definition of the s^* operation. We can check that $\sigma^* \upharpoonright_p : TA \Rightarrow B$ is just a copy of $\sigma \upharpoonright_p : A \Rightarrow B$, where all the moves on the left of the arrow are played in A_p (formally, we first need to prove by induction that $s^* \upharpoonright_p = (s \upharpoonright_p)^*$). Since $\sigma \upharpoonright_p$ is a strategy, then $\sigma^* \upharpoonright_p$ is also a strategy. □

Definition 2.97 (Concurrent interaction)**.** A *concurrent interaction* u on A, TB, TC, written $u \in \text{CInt}(A, TB, TC)$ is a pointed sequence on the arena $(A \Rightarrow TB) \Rightarrow TC$ such that $u \upharpoonright_{A,TB}$, $u \upharpoonright_{TB,TC}$ and $u \upharpoonright_{A,TC}$ are concurrent plays on $A \Rightarrow TB$, $TB \Rightarrow TC$ and $A \Rightarrow TC$, respectively.

Definition 2.98 (Concurrent composition)**.** Given two concurrent strategies $\sigma : A \Rightarrow TB$ and $\tau : B \Rightarrow TC$, their composition $\sigma;\tau : A \Rightarrow TC$ is defined as follows:

$$\sigma;\tau = \{u \upharpoonright_{A,TC} \mid u \in \text{CInt}(A, TB, TC) \text{ and } u \upharpoonright_{A,TB} \in \sigma \text{ and } u \upharpoonright_{TB,TC} \in \tau^*\}$$

Claim 2.99. *$\sigma;\tau$ is a concurrent strategy on $A \Rightarrow TC$.*

Proving this claim formally would require some technical game semantics machinery that we did not introduce. However, as we explain later in Remark 2.103, this is an instance of a more general construction, namely the composition in a Kleisli category, which should prove this claim as a consequence of the usual composition in $\mathbf{Str}$. We simply admit this claim for now.

Proposition 2.100. *We have a category $\mathbf{CStr}$ whose objects are arenas and morphisms from A to B are concurrent strategies on $A \Rightarrow TB$. The identity morphisms are given by $\eta_A : A \Rightarrow TA$, and the composition is the one defined above.*

In order to interpret our programming language in this category, we need one more operation, corresponding to running n sequential programs in parallel. Suppose given n strategies $(\sigma_p)_{p\in[n]}$ on the arena $A \Rightarrow B$. Then we can interleave them to get a concurrent strategy on $A \Rightarrow TB$:

Definition 2.101 (Shuffling)**.** Let $\sigma_p : A \Rightarrow B$ in $\mathbf{Str}$ for all $p \in [n]$. We define the concurrent strategy $F(\sigma_0, \dots, \sigma_{n-1}) : A \Rightarrow TB$ as follows:

$$F(\sigma_0, \dots, \sigma_{n-1}) = \{s \text{ pointed sequence on } A \Rightarrow TB \mid \forall p \in [n],\ s \upharpoonright_p \in \sigma_p\}$$

Proposition 2.102. *$F(\sigma_0, \dots, \sigma_{n-1})$ is a concurrent strategy on $A \Rightarrow TB$.*

Proof. The pointed sequences $s \in F(\sigma_0, \ldots, \sigma_{n-1})$ are concurrent plays since each component $s\restriction_p \in \sigma_p$ is a play. Moreover, $F(\sigma_0, \ldots, \sigma_{n-1})$ is closed under P-prefix since each of the strategies σ_p is closed under P-prefix. Finally, we can show that $F(\sigma_0, \ldots, \sigma_{n-1})\restriction_p = \sigma_p$, so it is a strategy on $A \Rightarrow B$. □

We can finally give a semantics to our protocols. Let $\mathcal{P}$ be a protocol, i.e., of the form

$$\begin{aligned} \mathcal{P} \quad = \quad & \mathsf{shared}\ a_1 : \tau_1, \ldots, a_k : \tau_k \\ & \mathsf{implements}\ \tau \\ & P_0 \parallel P_1 \parallel \ldots \parallel P_{n-1} \end{aligned}$$

The usual sequential game semantics allow us to associate a strategy in the category $\mathbf{Str}$ with each program P_i, which we write $[\![P_i]\!] : [\![\tau_1]\!] \times \ldots \times [\![\tau_k]\!] \Rightarrow [\![\tau]\!]$. Then, the semantics of $\mathcal{P}$ is $[\![\mathcal{P}]\!] := F([\![P_0]\!], \ldots, [\![P_{n-1}]\!])$, which is a concurrent strategy on the arena $[\![\tau_1]\!] \times \ldots \times [\![\tau_k]\!] \Rightarrow T[\![\tau]\!]$. Note that each object type τ_i is interpreted using the call-by-value arrow, for instance, the interpretation of the type of read/write registers is $[\![(\mathsf{Unit} \to \mathsf{Nat}) \times (\mathsf{Nat} \to \mathsf{Unit})]\!] := (\mathbf{U} \overset{\text{cbv}}{\Longrightarrow} \mathbf{N}) \times (\mathbf{N} \overset{\text{cbv}}{\Longrightarrow} \mathbf{U})$.

Remark 2.103. As our notations and terminology suggests, these definitions of concurrent strategies are not just an ad-hoc constructions, but an instantiation of Moggi's categorical semantics for computational effects using *monads* [95, 96]. The operator $TA = A^n$ on arenas can actually be extended to a monad in the category of strategies, with η_A and σ^* playing the role of the return and bind operations. The composition of the category $\mathbf{CStr}$ is defined similarly to the one of the Kleisli category over T; however, one must be a bit careful. In the Kleili category $\mathbf{Str}_T$, the strategies are globally alternating, which is stronger than the component-wise alternation that we impose on $\mathbf{CStr}$. In fact, what we must do is define the monad T in the category $\mathbf{NStr}$ of non-alternating strategies. Then, $\mathbf{CStr}$ can be defined as a subcategory of the Kleisli category $\mathbf{NStr}_T$. This more categorical approach will be developed in a future paper.

Future work. This game semantics approach for fault-tolerant protocols is still work in progress. We have the basic definitions in place, but many questions remain to be investigated.

First, the link with our trace-based computational model of Sections 2.2 and 2.3 should be clarified. The notion of concurrent specifications of Definition 2.22 (say, with input and output values $\mathcal{V} = \mathbb{N}$) should closely correspond to concurrent strategies on the arena $\mathbf{1} \Rightarrow T(\mathbf{N} \overset{\text{cbv}}{\Longrightarrow} \mathbf{N})$, with one additional restriction which is the expansion property. Then, in Section 2.3.2, in order to define the semantics of protocols, we first define the set $\mathcal{T}_P$ of *valid executions* of our protocol $\mathcal{P}$, by considering traces with inner and outer actions. This set $\mathcal{T}_P$ should correspond to a strict subset of the concurrent strategy $\sigma_{\mathcal{P}}^* : TA \Rightarrow TB$, where $\sigma_{\mathcal{P}} : A \Rightarrow TB$ is the concurrent strategy associated to $\mathcal{P}$ in our game semantics. Indeed, in $\mathcal{T}_P$ we are assuming that the shared objects behave according to their specification, whereas in game semantics they are unspecified. Finally, in Definition 2.46, we remove the inner actions from $\mathcal{T}_P$ to obtains the semantics $[\![\mathcal{P}]\!]$ of the protocol, which is a concurrent specification. In game semantics, this corresponds to precomposing the strategy $\sigma_{\mathcal{P}} : A \Rightarrow TB$ by a strategy $\tau : \mathbf{1} \Rightarrow TA$, specifying the behavior of the object. We obtain a strategy $\tau; \sigma_{\mathcal{P}} : \mathbf{1} \Rightarrow TB$, which should correspond to $[\![P]\!]$.

As we mentioned in Remark 2.89, an important question in game semantics is whether our model is *fully abstract*. In our case, a natural idea seems to be to extend the usual conditions on strategies (deterministic, total, innocent, well-bracketed, ...) to concurrent strategies in a component-wise manner.

Namely, a concurrent strategy $\sigma : A \Rightarrow TB$ is *deterministic* / *total* / *innocent* if for every process p, the p-restriction $\sigma\restriction_p$ is deterministic / total / innocent / etc. Is this sufficient to get completeness and full abstraction? It might not be the case, since some of these notions are known to be problematic in the presence of asynchrony [92].

Finally, an important result of game semantics that might be useful in the context of fault-tolerant distributed computing is the characterization of strategies with memory [1]. By weakening the innocence condition, one can identify precisely the strategies that arise when a sequential program has access to a global memory. Our goal would be to transpose this result to the setting of concurrent processes communicating through shared read/write registers. Having an exact characterization of the strategies that can be produced in this setting might give us new tools to analyze the solvability of concurrent tasks in a read/write shared memory setting.

CHAPTER 3

Epistemic logic semantics

From the very beginning of the topological approach for fault-tolerant computability, the notions of *knowledge* and *indistinguishability* have been used to give an intuitive explanation of the constructions at hand. A very revealing example is the 1993 paper of Saks and Zaharoglou is titled: "*Wait-free K-set Agreement is Impossible: The Topology of Public Knowledge*". Indeed, in the simplicial complex which is nowadays known as the *protocol complex*, the facets are interpreted as executions of the protocol, while vertices represent the local view of a single process in such an execution. When a given vertex belongs to two facets, this means that the corresponding process cannot *distinguish* between the two executions. In other words, the process does not *know* which of the two executions occurred, and therefore it must choose the same decision value in both situations.

These notions of indistinguishability and knowledge are central in the field of *epistemic logic*, the modal logic of knowledge. Epistemic logic is concerned with reasoning about systems where a finite number of agents can know facts about the world, and about what the other agents know. The usual model for multi-agent epistemic logic is based on a *Kripke frame*, which is a graph whose vertices represent the possible worlds, and edges are labeled with agents that do not distinguish between two worlds. Such a Kripke model represents the knowledge of the agents about a given situation. Our central result is the following observation: the information contained in a Kripke model can be reformulated using the formalism of chromatic simplicial complexes, to obtain a new kind of model for epistemic logic called *simplicial models*. Epistemic logic formulas can be interpreted in those simplicial models, while preserving the nice properties of Kripke models, such as soundness and completeness with respect to (a slightly modified version of) the axiomatic system $\mathbf{S5_n}$. Moreover, we prove that these simplicial models are very closely related to the usual Kripke models: there is an equivalence of categories between the two structures. Thus, we are merely uncovering the higher-dimensional topological information which is already present in Kripke models.

To explain the interest of this new kind of models, we use them to study fault-tolerant distributed computability, because its intimate relation with topology is well understood [64]. However, epistemic logic is a static study of an epistemic situation; while distributed computing is concerned with how the knowledge of the processes evolves during the computation. The field which studies how the knowledge of the agents evolve when communication occurs is called *Dynamic Epistemic Logic (DEL)* [7, 27]. More precisely, the variant of DEL that we use is based on the notion of *action model* [8, 6], which specifies the communicative *actions* that might occur, along with their effect on the system. The key notion of DEL is the so-called *product-update* operation: given an initial Kripke model and an action model, we can combine them to obtain a new Kripke model, representing the knowledge of the agents after an action has occurred. We extend the equivalence between simplicial models and Kripke models to the context of DEL, by defining a simplicial version the product-update operation. When modeling distributed computing, this product-update operation will play the role of the carrier maps that we used in Chapter 1.

This connection between epistemic logic and distributed computability has important consequences for both fields. From the point of view of epistemic logic, uncovering the higher-dimensional topological structure hidden in Kripke models allows us to transport methods that have been used successfully in distributed computability [64] to the realm of DEL. In particular, the knowledge gained by applying an action model is intimately related to how it modifies the topology of the initial model. Conversely, the benefit of this approach in distributed computing is in providing a formal epistemic logic semantics to task solvability. The abstract topological arguments that are used to prove impossibility results can now be given a concrete interpretation in terms of how much knowledge is necessary to solve a task.

Most of the results presented in this chapter were published in two papers: GandALF 2018 [53] and DaLi 2019 [51]. Some preliminary versions of these findings appear in two Technical Reports [46, 47].

Related work. Work on knowledge and distributed systems is of course one of the main inspirations of the present work [106, 58], especially where connectedness [17, 19, 29] is used. To the best of our knowledge, no previous work uses Dynamic Epistemic Logic [7, 27] to study such systems, and neither on directly connecting the combinatorial topological setting of [64] with Kripke models. One approach worth mentioning is a categorical connection between Kripke frames and geometry [104, 103] that appears in the context of multiagent systems. In [72], the author proposes a variant of (non-dynamic) epistemic logic for a restricted form of wait-free task specification that cannot account for important tasks such as consensus. Similar to [97], we show that even though a problem may not explicitly mention the agents' knowledge, it can in fact be restated as knowledge gain requirements. Nevertheless, we exploit the “runs and systems" framework in an orthogonal way, and the knowledge requirements we obtain are about inputs; common knowledge in the case of consensus, but other forms of nested knowledge for other tasks. In contrast, the *knowledge of precondition principle* of [97] implies that common knowledge is a necessary condition for performing simultaneous actions [58].

DEL is often thought of as inherently being capable of modeling only agents that are synchronous, but as discussed in [25], this is not the case. More recently, [78] proposes a variant of public announcement logic for asynchronous systems that introduces two different modal operators for sending and receiving messages. A similar approach of asynchronous announcement has been taken in [116], furthermore showing that on multi-agent epistemic models, each formula in asynchronous announcement logic is equivalent to a formula in epistemic logic. We show here that DEL can naturally model the knowledge in an asynchronous distributed system, at least as far as it is concerned with task solvability.

Finally, note that our formulation of carrier maps as products has been partially observed in [59].

Plan. The chapter is organized as follows. We start by recalling in Section 3.1 the usual notion of model for epistemic logic, based on Kripke frames. Section 3.2 introduces our new notion of model based on chromatic simplicial complexes, and prove that they are equivalent to Kripke models. In Section 3.3, we define the distributed computing notions of protocols, tasks and solvability using our DEL framework. Then, in Section 3.4, we analyze several examples of tasks in order to showcase the benefits of the DEL approach to study the solvability of concurrent tasks. In Section 3.5 we study yet another example, but this time we show that our proof method is unable to prove the unsolvability of the task. We find a workaround to this issue in Section 3.6, and discuss its relevance. Finally in Section 3.7, we conclude and discuss future research directions.

3.1 Preliminaries

We start by recalling the usual way in which (dynamic) epistemic logic is studied in the literature, using Kripke models. Therefore, this section only contains standard definitions of epistemic logic, which can be found for example in [7, 27]. For a very thorough study of modal logics in general, see [11]. We will revisit those notions in Section 3.2 using a topological approach.

3.1.1 Syntax

Epistemic logic is the modal logic of *knowledge*. Its syntax is based on standard propositional logic, to which we add modal operators. The most important one is the *knowledge operator*, which is written $K_a\varphi$ and read "a knows φ". It corresponds to the usual $\Box$ operator from modal logic, and is moreover parameterized by an *agent* a, taken from some finite set. Intuitively, the agents are entities which are able to know things about the world, and about what the other agents know.

Let At be a countable set of *atomic propositions* and Ag a finite set of *agents*. The language of formulas $\mathcal{L}_K(\mathrm{Ag}, \mathrm{At})$, or just $\mathcal{L}_K$ if Ag and At are implicit, is generated by the following BNF grammar:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_a\varphi \qquad p \in \mathrm{At},\ a \in \mathrm{Ag}$$

As usual, can derive other logical connectives from these, such as the disjunction $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$ and the implication $\varphi \Rightarrow \psi := \neg\varphi \vee \psi$. For a set $A \subseteq \mathrm{Ag}$ of agents, we define the *everybody knows* operator as $E_A\varphi = \bigwedge_{a\in A} K_a\varphi$. For example, $E_{\{a,b\}}\varphi$ means that "both a and b know φ". The dual of the K_a operator, $\neg K_a\neg\varphi$, intuitively means that "a considers it possible that φ"; but we do not introduce a notation for it since we do not use it in the rest of the chapter.

Another important operator of epistemic logic is called *common knowledge*. It is not definable using the language $\mathcal{L}_K$, so we need to extend it. We denote by $\mathcal{L}_{CK}$ the language generated by the grammar:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_a\varphi \mid C_A\varphi \qquad p \in \mathrm{At},\ a \in \mathrm{Ag}, A \subseteq \mathrm{Ag}$$

For a group of agents $A \subseteq \mathrm{Ag}$, common knowledge of φ among A is, semantically, the greatest solution of the equation $C_A\varphi = \varphi \wedge E_A(C_A\varphi)$. In other words, the formula $C_A\varphi$ of $\mathcal{L}_{CK}$ behaves like an infinite formula $C_A\varphi = \varphi \wedge E_A\varphi \wedge E_AE_A\varphi \wedge \ldots$ saying that the formula φ is true and moreover, among the set of agents A, everyone knows φ, and everyone knows that everyone knows φ, and so on.

The logic $\mathbf{S5_n}$

Studying the proof theory of epistemic logic is not at all our main interest; but since we briefly mention some soundness and completeness result for $\mathbf{S5_n}$ in Section 3.2.3, let us define that logic. The deduction rules of $\mathbf{S5_n}$ are all the standard rules of propositional logic, along with the following axiom schemas:

– Axiom K: $K_a(\varphi \Rightarrow \psi) \Rightarrow (K_a\varphi \Rightarrow K_a\psi)$,
– Axiom T, or axiom of truth: $K_a\varphi \Rightarrow \varphi$,

– Axiom 5, or negative introspection: $\neg K_a\varphi \Rightarrow K_a\neg K_a\varphi$,
– Rule of necessitation: if φ is derivable without assumption, then so is $K_a\varphi$.

Note that the rule of necessitation is *not* the axiom schema $\varphi \Rightarrow K_a\varphi$ ("the agents know everything which is true"); it merely says that the agents are "rational", in the sense that they are able to make logical deductions. The deduction rules of $\mathbf{S5_n}$ are summed up below. For the propositional fragment, we use the rules of natural deduction, written using the sequent notation $\Gamma \vdash \varphi$, where Γ is a set of formulas. Intuitively, such a sequent means that the formula φ is provable using the set of hypothesis Γ.

$$\overset{\textsc{Hyp}}{\frac{}{\Gamma, \varphi \vdash \varphi}} \qquad \overset{\textsc{Ex Falso}}{\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}} \qquad \overset{\neg\textsc{Intro}}{\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}} \qquad \overset{\neg\textsc{Elim}}{\frac{\Gamma \vdash \varphi \wedge \neg\varphi}{\Gamma \vdash \bot}} \qquad \overset{\wedge\textsc{Intro}}{\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}} \qquad \overset{\wedge\textsc{Elim}_i}{\frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_i}}$$

$$\overset{\text{EM}}{\frac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}} \qquad \overset{\text{K}}{\frac{\Gamma \vdash K_a(\varphi \Rightarrow \psi) \qquad \Gamma \vdash K_a\varphi}{\Gamma \vdash K_a\psi}} \qquad \overset{\text{T}}{\frac{\Gamma \vdash K_a\varphi}{\Gamma \vdash \varphi}} \qquad \overset{5}{\frac{\Gamma \vdash \neg K_a\varphi}{\Gamma \vdash K_a\neg K_a\varphi}} \qquad \overset{\textsc{Nec}}{\frac{\vdash \varphi}{\vdash K_a\varphi}}$$

Note that the standard rules for the derived connectives, such as modus ponens, are admissible in this system. Another important rule which is also admissible in this system (cf. [110]) is the so-called *Axiom 4*, which says that $K_a\varphi \Rightarrow K_aK_a\varphi$.

$$\overset{\textsc{Modus Ponens}}{\frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}} \qquad \overset{4}{\frac{\Gamma \vdash K_a\varphi}{\Gamma \vdash K_aK_a\varphi}}$$

If we include the common knowledge operator, the corresponding logic is called $\mathbf{S5C_n}$. It is obtained by adding to $\mathbf{S5_n}$ the following deduction rules:

– Axiom K for C_A: $C_A(\varphi \Rightarrow \psi) \Rightarrow (C_A\varphi \Rightarrow C_A\psi)$,
– Mix axiom: $C_A\varphi \Rightarrow (\varphi \wedge E_AC_A\varphi)$,
– Induction axiom: $C_A(\varphi \Rightarrow E_A\varphi) \Rightarrow (\varphi \Rightarrow C_A\varphi)$,
– Necessitation of C_A: if φ is derivable without assumption, then so is $C_A\varphi$.

3.1.2 Kripke semantics

Kripke frames. Modal logics are usually interpreted using variants of *Kripke frames*, that is, sets of *possible worlds* equipped with a binary relation. Different flavors of modal logic can be modeled by imposing various conditions on this binary relation. In the case of multi-agent epistemic logic $\mathbf{S5_n}$, the relevant notion of model is based on Kripke frames with one binary relation for each agent, and such that each of them is an equivalence relation. Since it is the only kind of frame that we will use, we simply call them "Kripke frames".

Definition 3.1. A *Kripke frame* $M = \langle W, (\sim_a)_{a\in\mathrm{Ag}}\rangle$ over a set Ag of agents consists of a set of worlds W and a family of equivalence relations on W, written $\sim_a \in W \times W$ for every $a \in \mathrm{Ag}$. Two worlds $v, w \in W$ such that $v \sim_a w$ are said to be *indistinguishable* by a. When the set of agents is clear from the context, we usually write $M = \langle W, \sim\rangle$.

A Kripke frame is *proper* if any two worlds can be distinguished by at least one agent. Thus, being proper means that the intersection of all equivalence relations $\sim_a$ is the identity; in other words, the *distributed knowledge* (see e.g. [106]) of all the agents together is enough to identify a world uniquely. In the remainder of the chapter, we will focus on proper frames. This is not done without loss of generality; the consequences of this assumption will be discussed later on. Let $M = \langle W, \sim \rangle$ and $M' = \langle W', \sim' \rangle$ be two Kripke frames. A *morphism* from M to M' is a function f from W to W' such that for all $v, w \in W$, for all $a \in \mathrm{Ag}$, $v \sim_a w$ implies $f(v) \sim'_a f(w)$. We write $\mathcal{K}_{\mathrm{Ag}}$ for the category of proper Kripke frames, with morphisms of Kripke frames as arrows.

Kripke models. To get a notion of model, we simply decorate each world with a set of atomic propositions, which correspond to the propositions which are true in this particular world.

Definition 3.2. A *Kripke model* $M = \langle W, \sim, L \rangle$ consists of a Kripke frame $\langle W, \sim \rangle$ and a labeling $L : W \to \mathscr{P}(\mathrm{At})$. Intuitively, $L(w)$ is the set of atomic propositions that are true in the world w.

A Kripke model is *proper* if the underlying Kripke frame is proper. Let $M = \langle W, \sim, L \rangle$ and $M' = \langle W', \sim', L' \rangle$ be two Kripke models on the same set At of atomic propositions. A *morphism* of Kripke models $f : M \to M'$ is a morphism of the underlying Kripke frames such that $L'(f(s)) = L(s)$ for every world w in W. We write $\mathcal{KM}_{\mathrm{Ag,At}}$ for the category of proper Kripke models with such morphisms.

An epistemic logic formula $\varphi \in \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ is interpreted in one world of a Kripke model. We write $M, w \models \varphi$ to denote that the formula φ is true in the world w of the Kripke model M.

Definition 3.3. Given a Kripke model $M = \langle W, \sim, L \rangle$ and a world $w \in W$, we define the *truth* of a formula $\varphi \in \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ in the world w by induction on φ as follows.

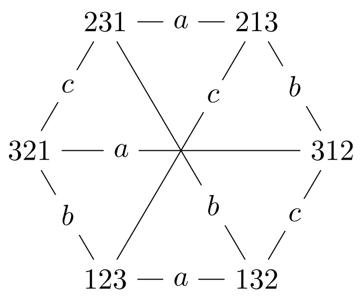
$$\begin{array}{lll}
M, w \models p & \text{if} & p \in L(w) \\
M, w \models \neg\varphi & \text{if} & M, w \not\models \varphi \\
M, w \models \varphi \wedge \psi & \text{if} & M, w \models \varphi \text{ and } M, w \models \psi \\
M, w \models K_a\varphi & \text{if} & \text{for all } w' \in W, w \sim_a w' \text{ implies } M, w' \models \varphi \\
M, w \models C_A\varphi & \text{if} & \text{for all } w' \in W \text{ in the } A\text{-connected component of } w,\ M, w' \models \varphi
\end{array}$$

Here, by "A-connected component", we mean that w and w' are connected through a finite sequence of worlds $w \sim_{a_1} w_1 \sim_{a_2} \cdots \sim_{a_k} w_k \sim_{a_{k+1}} w'$, where $a_i \in A$ for every agent a_i in the sequence.

In the previous definition, the first three cases are just Tarski's usual definition of truth for propositional logic. The fourth case is the most interesting one: it says that an agent $a \in \mathrm{Ag}$ *knows* a formula φ in some world w whenever that formula is true in every possible world w' which is indistinguishable from w by that agent. Finally, the last case, for common knowledge, is obtained by iterating the previous one any number of times. If $M, w \models \varphi$ holds for every world $w \in W$, we write $M \models \varphi$; and we write $\models \varphi$ if $M \models \varphi$ holds for every Kripke model M.

Example 3.4. Consider the following scenario. There are three agents, a, b, and c, and a deck of three cards, $\{1, 2, 3\}$. Each agent is dealt a card from the deck, face down, and then the agents are allowed to look at the value of their own card. Let us describe the Kripke model $M = \langle W, \sim, L \rangle$ representing that situation. There are six possible worlds, corresponding to the six possible assignments of cards to agents. We write the set of worlds $W = \{123, 132, 213, 231, 312, 321\}$, where the three digits correspond to the card given to a, b, c, respectively. The indistinguishability relation models the fact that an agent only knows its own card: thus, worlds where the first digit are the same are indistinguishable for agent a, i.e., $123 \sim_a 132$ and $213 \sim_a 231$ and $312 \sim_a 321$; and similarly for $\sim_b$ and $\sim_c$, using the second and third digits. We choose the set of atomic propositions $\mathrm{At} = \{\mathrm{has}_i(x) \mid i \in \{1, 2, 3\} \text{ and } x \in \{a, b, c\}\}$. Intuitively, the proposition $\mathrm{has}_i(x)$ means that agent x has card i. The worlds are labeled by atomic propositions as expected: for example, $L(123) = \{\mathrm{has}_1(a), \mathrm{has}_2(b), \mathrm{has}_3(c)\}$.

This Kripke model is represented below as a graph, where the worlds are the vertices, and the $\sim$ relations are represented as labeled edges. The labeling of atomic propositions is not depicted.



For example, in the world 123, the formulas $\mathrm{has}_1(a)$ and $K_a(\mathrm{has}_1(a))$ are true; the formula $K_b(\mathrm{has}_1(a))$ is false; the formula $K_a\neg K_b(\mathrm{has}_1(a))$ is true; the formula $K_a(\mathrm{has}_2(b) \vee \mathrm{has}_3(b))$ is true; and the formula $C_{\{a,b,c\}}(\mathrm{has}_1(a) \vee \mathrm{has}_1(b) \vee \mathrm{has}_1(c))$ is true.

It is well-known that the axiom systems $\mathbf{S5_n}$ and $\mathbf{S5C_n}$ are *sound and complete* for the languages $\mathcal{L}_K$ and $\mathcal{L}_{CK}$, with respect to the class of Kripke models [27], i.e.,

Theorem 3.5 (Soundness and completeness)**.** *For every formula $\varphi \in \mathcal{L}_K(\mathrm{Ag}, \mathrm{At})$, we have*

$$\models \varphi \quad \iff \quad \vdash_{\mathbf{S5_n}} \varphi$$

and for every formula $\varphi \in \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$,

$$\models \varphi \quad \iff \quad \vdash_{\mathbf{S5C_n}} \varphi$$

3.1.3 Dynamic Epistemic Logic

Dynamic Epistemic Logic (DEL) [7, 27] is a family of modal logics which study how an epistemic model changes when the agents learn new information about the world. The syntax of epistemic logic is enriched by a new modal operator $[\alpha]\varphi$, which expresses that the formula φ is true after some *action* α has occurred. An action can be thought of as an announcement made by the environment, which is not necessarily public, in the sense that not all agents receive these announcements. Semantically, to determine whether the formula $[\alpha]\varphi$ is true in some world w of a Kripke model M, we construct a new Kripke model $M[\alpha]$ and world $w[\alpha]$, representing the new knowledge of the agents after the action has occurred, and we investigate whether φ is true in this new model. There are many variants of DEL, depending on what kind of actions we want to consider: public announcements, communications between agents with various degrees of privacy, lying, forgetting, and so on. One variant of DEL which can express various complex epistemic interactions is the one based on *action models* [6]. An action model is a relational structure that can be used to describe a variety of informational actions. This action model describes all the possible actions that might happen, as well as how they affect the different agents.

We do not introduce formally the syntax and semantics of DEL formulas, because we will not be using them. In fact, the key notion of DEL that we are interested in is the so-called *product-update* operation: given an epistemic model M and an action model $\mathcal{A}$, the product update $M[\mathcal{A}]$ is a new model which describes all the new possible worlds after an action from $\mathcal{A}$ has occurred in M. Suppose fixed a set Ag of agents and a set At of atomic propositions.

Definition 3.6. An *action model* is a structure $\mathcal{A} = \langle T, \sim, \mathsf{pre}\rangle$, where T is a set of *actions*, such that for each $a \in \mathrm{Ag}$, $\sim_a$ is an equivalence relation on T called the indistinguishability relation, and $\mathsf{pre} : T \to \mathcal{L}_K(\mathrm{Ag}, \mathrm{At})$ is a function that assigns a *precondition* formula $\mathsf{pre}(t)$ to each action $t \in T$.

Intuitively, given an initial model M and an action $t \in T$, the role of the precondition formula $\mathsf{pre}(t) \in \mathcal{L}_K(\mathrm{Ag}, \mathrm{At})$ is to select a subset of the worlds of M, which consists of the worlds where $\mathsf{pre}(t)$ is true. These worlds are the ones where the action t is allowed to occur. Moreover, some agent a may not be able to distinguish some actions t and t'; in which case, if action t occurs, the agent a only learns the fact that one action in the $\sim_a$-equivalence class of t has occurred.

Definition 3.7 (Product update)**.** Let $M = \langle W, \sim, L\rangle$ be an initial Kripke model, and $\mathcal{A} = \langle T, \sim, \mathsf{pre}\rangle$ an action model. The *product-update model* $M[\mathcal{A}]$ is defined as $M[\mathcal{A}] = \langle W[\mathcal{A}], \sim^{[\mathcal{A}]}, L[\mathcal{A}]\rangle$, where each world of $W[\mathcal{A}]$ is a pair (w, t) with $w \in W,\ t \in T$ such that $\mathsf{pre}(t)$ holds in w, i.e., $M, w \models \mathsf{pre}(t)$. Then, we define $(w, t) \sim_a^{[\mathcal{A}]} (w', t')$ whenever both $w \sim_a w'$ and $t \sim_a t'$. The valuation $L[\mathcal{A}]$ at a pair (w, t) is just as it was at w, i.e., $L[\mathcal{A}]((w, t)) = L(w)$.

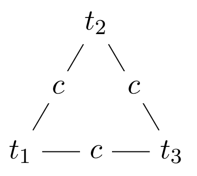
It is easily shown that $\sim^{[\mathcal{A}]}$ is an equivalence relation, and therefore $M[\mathcal{A}]$ is a Kripke model. Intuitively, the world (w, t) of $M[\mathcal{A}]$ represents the situation where we started from the world w of M and the action t occurred. In order to be able to distinguish between two such worlds (w, t) and (w', t'), an agent must be able to either distinguish the world w from w', or to distinguish the action t from t'.

Example 3.8. We start from the initial Kripke model M of Example 3.4, with three agents $\mathrm{Ag} = \{a, b, c\}$ and three cards $\{1, 2, 3\}$. Let us model the situation where agent a publicly reveals its card to the others. There are three possible actions: a reveals that it has card 1; a reveals that it has card 2; and a reveals that it has card 3. We name these three actions t_1, t_2 and t_3, and let $T = \{t_1, t_2, t_3\}$. Since the announcement is public, everyone is able to distinguish between the actions. So, we let $\sim_x$ be the identity relation for all $x \in \mathrm{Ag}$. Finally, the preconditions indicate that a can reveal only its own card: $\mathsf{pre}(t_1) = \mathrm{has}_1(a)$, and similarly for t_2 and t_3. If we write $\mathcal{A} = \langle T, \sim, \mathsf{pre}\rangle$, the product update model $M[\mathcal{A}]$ is depicted below.

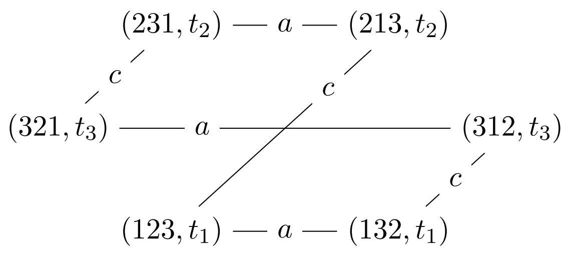
$$\begin{tikzcd}
(231, t_2) \arrow[r, dash, "a"] & (213, t_2) \\
(321, t_3) \arrow[r, dash, "a"] & (312, t_3) \\
(123, t_1) \arrow[r, dash, "a"] & (132, t_1)
\end{tikzcd}$$

Assume that we started in the world 123 of M, and action t_1 occurred; then we arrive in the world $(123, t_1)$ of $M[\mathcal{A}]$. In this world, the formula $E_{\{b,c\}}(\mathrm{has}_1(a) \wedge \mathrm{has}_2(b) \wedge \mathrm{has}_3(c))$ is true, that is, both b and c know all the cards of the other agents. The agent a still does not know the cards of the others ($K_a \, \mathrm{has}_2(b)$ is false), but it did learn some new information: for instance, now the formula $K_a K_b \, \mathrm{has}_1(a)$ is true.

Example 3.9. We once again start from the same initial model M of Example 3.4, but now agent a reveals its card privately to agent b. Agent c sees the communication happen, but does not see the card being revealed. The action model $\mathcal{A} = \langle T, \sim, \mathsf{pre} \rangle$ is represented below, on the left. The three actions t_1, t_2, t_3 are the same as in the previous example, with the same preconditions, but now agent c cannot distinguish between them.



Action model $\mathcal{A}$



Product update model $M[\mathcal{A}]$

If we start once again in the world 123 and action t_1 occurs, we can see that agent c has learned something from this interaction. Indeed, the formula $K_c(\mathrm{has}_1(a) \Rightarrow K_b \, \mathrm{has}_1(a))$ is true in the world $(123, t_1)$ of $M[\mathcal{A}]$, whereas it was false in the world 123 of M.

Example 3.10. To model a situation where a reveals its card to b in a non-public way, such that agent c considers it possible that nothing happened, we can proceed as follows. We add one more action, which means intuitively that nothing happened. So, the set of actions is $T = \{t_1, t_2, t_3, n\}$. The agent c cannot distinguish between any of those actions; and the agents a and b can distinguish between all of them. The precondition of the action n is a tautology: so, n can happen in any world.

The product update model $M[\mathcal{A}]$ has 12 possible worlds. It contains a copy of M (containing the worlds of the form (w, n)), a copy of the product update model of Example 3.9 (containing the worlds of the form (w, t_i)), and some c-labeled edges between the two copies. It can be checked that in the world $(123, t_1)$ of this model, the agent c has learned nothing: the true formulas of the form $K_c \, \varphi$ are the same as in M.

3.2 Simplicial complex models for Dynamic Epistemic Logic

We describe here a new kind of model for epistemic logic, based on chromatic simplicial complexes. We will show that these simplicial models are very closely related to the usual Kripke models: there is an equivalence of categories between the two structures. This means that our models are merely a reformulation of Kripke models using a different formalism. The geometric nature of simplicial complexes allows us to consider higher-dimensional topological properties of our models, and investigate their meaning in terms of knowledge. Of course, the idea of using simplicial complexes comes from the topological approach to distributed computability, presented in Chapter 1. The link between DEL and distributed computing will be developed in the next sections.

We begin in Section 3.2.1 by defining our notion of simplicial models for epistemic logic and the associated semantics. We prove the equivalence of categories between simplicial models and Kripke models in Section 3.2.2. This equivalence works under a *locality* restriction on Kripke models, but as we show in Section 3.2.4, this assumption can be lifted for finite models. In Section 3.2.3 we show that our class of models is sound and complete w.r.t. the logics $\mathbf{S5_n}$ and $\mathbf{S5C_n}$. Finally, in Section 3.2.5, we define a simplicial product update operation, allowing us to use our simplicial models in the context of DEL.

3.2.1 Simplicial models

Recall from Section 1.2 the notions of chromatic simplicial complexes and chromatic simplicial maps. In the following, we work with $n+1$ agents, and write $\mathrm{Ag} = \{a_0, \dots, a_n\}$ for the set of agents. From now on, the agents will sometimes be regarded as colors.

For technical reasons, we restrict to models where all the atomic propositions are saying something about some local value held by one particular agent. All the examples that we are interested in will fit in that framework (that was for example the case in Example 3.4). Let $\mathcal{V}$ be some countable set of values, and $\mathrm{At} = \{p_{a,x} \mid a \in \mathrm{Ag}, x \in \mathcal{V}\}$ be the set of *atomic propositions*. Intuitively, $p_{a,x}$ is true if agent a holds the value x. In all our examples, an agent will hold exactly one value, but we do not enforce that in general. We write At_a for the atomic propositions concerning agent a. Thus, $\mathrm{At} = \bigcup_{a \in \mathrm{Ag}} \mathrm{At}_a$.

Definition 3.11. A *simplicial model* $M = \langle V, S, \chi, \ell \rangle$ consists of a pure chromatic simplicial complex $\langle V, S, \chi \rangle$ of dimension n, colored by the agents, equipped with a labeling $\ell : V \to \mathscr{P}(\mathrm{At})$ that associates with each vertex $v \in V$ a set of atomic propositions concerning agent $\chi(v)$, i.e., such that $\ell(v) \subseteq \mathrm{At}_{\chi(v)}$.

Given a simplex $X \in S$, we write $\ell(X) = \bigcup_{v \in X} \ell(v)$. A *morphism* of simplicial models $f : M \to M'$ is a chromatic simplicial map that preserves the labeling: $\ell'(f(v)) = \ell(v)$. We denote by $\mathcal{SM}_{\mathrm{Ag},\mathrm{At}}$ the category of simplicial models over the set of agents Ag and atomic propositions At. Intuitively, each *facet* (i.e., n-simplex) of M corresponds to one possible world of the model. This facet has $n+1$ vertices, one for each agent, so that whenever an agent cannot distinguish between two worlds, the two facets will be glued along the corresponding vertices. In the following, $\mathcal{F}(M)$ denotes the set of facets of M.

Given a simplicial model M, the truth of an epistemic formula is defined in one facet $X \in \mathcal{F}(M)$. The following definition is a direct translation of Definition 3.3, under the equivalence between simplicial models and Kripke models which will be explained in Section 3.2.2, Theorem 3.20.

Definition 3.12. The *truth* of a formula $\varphi \in \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ in some epistemic state (M, X) with $M = \langle V, S, \chi, \ell \rangle$ a simplicial model and $X \in \mathcal{F}(M)$ a facet of M, is written $M, X \models \varphi$. We define it by induction on φ as follows.

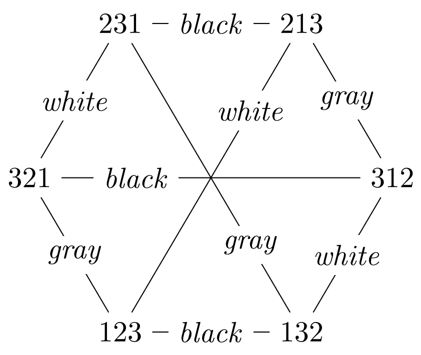
$$\begin{array}{lll}
M, X \models p & \text{if} & p \in \ell(X) \\
M, X \models \neg\varphi & \text{if} & M, X \not\models \varphi \\
M, X \models \varphi \wedge \psi & \text{if} & M, X \models \varphi \text{ and } M, X \models \psi \\
M, X \models K_a\varphi & \text{if} & \text{for all } Y \in \mathcal{F}(C), a \in \chi(X \cap Y) \text{ implies } M, Y \models \varphi \\
M, X \models C_A\varphi & \text{if} & \text{for all } Y \in \mathcal{F}(C) \text{ in the } A\text{-connected component of } X,\ M, Y \models \varphi
\end{array}$$

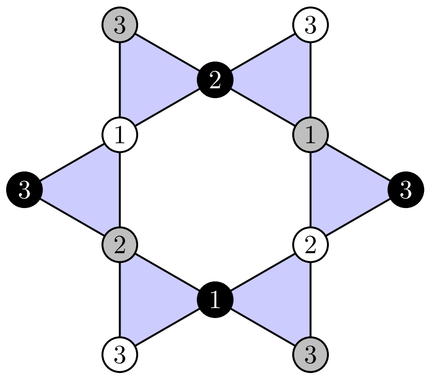
Here, by "A-connected component", we mean that X and Y are connected through a finite sequence of facets $X = X_0, X_1, \dots, X_k = Y$, where for all i, there is an agent $a \in A$ such that $a \in \chi(X_i \cap X_{i+1})$.

Example 3.13. We now model the epistemic situation of Example 3.4 as a simplicial model. The three agents will now be called $\mathrm{Ag} = \{\mathit{black}, \mathit{gray}, \mathit{white}\}$, and represented as colors on the pictures. The simplicial model $M = \langle V, S, \chi, \ell \rangle$ is defined as follows.

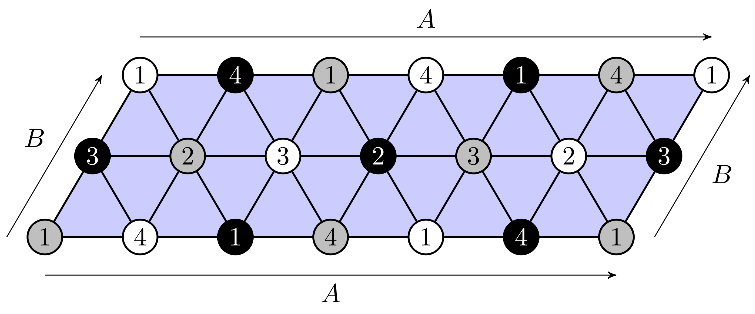
– There are 9 vertices, one for each pair of agent and card, i.e., $V = \mathrm{Ag} \times \{1, 2, 3\}$.
– There are six facets, corresponding to the six ways to distribute one card to each agent. The set S of simplices consists of all the faces of these facets.
– The vertex $(a, i) \in V$ is colored by a and labeled by $\{\mathrm{has}_i(a)\}$.

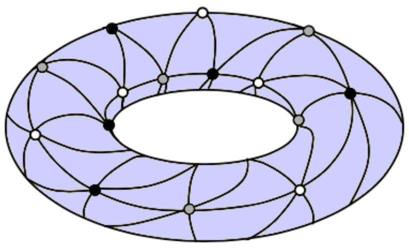
This simplicial complex is represented below (right). In the planar drawing, vertices that appear several times with the same color and same card should be identified. To visualize the real simplicial model, one should fold each pair of opposite triangles towards the inside, to join the two vertices with the same card and color. For comparison, the Kripke model of Example 3.4 is represented on the left, with renamed agents.





Example 3.14. We now consider the same situation as in the previous example, but now there are four possible cards $\{1, 2, 3, 4\}$. The three agents *black*, *gray* and *white* are given one card each from the deck, and the remaining card is kept hidden. So, each agent knows only its own card. A planar representation of the resulting simplicial model is depicted below, on the left. As before, some vertices have been duplicated and should be identified. In fact, the two arrows A and B indicate how the edges should be glued together. This gluing diagram is well known to topologists: what we obtain is a triangulated torus.





The following lemma is a classic result of modal logics, saying that morphisms of simplicial models cannot "gain knowledge about the world". This will be useful in Section 3.3.3 when we formulate the solvability of a task as the existence of some morphism.

Lemma 3.15 (knowledge gain). *Consider simplicial models $M = \langle V, S, \chi, \ell\rangle$ and $M' = \langle V', S', \chi', \ell'\rangle$, and a morphism $f : M \to M'$. Let $X \in \mathcal{F}(M)$ be a facet of M, a an agent, and $\varphi \in \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ a "positive" formula, i.e. which does not contain negations except, possibly, in front of atomic propositions. Then, $M', f(X) \models \varphi$ implies $M, X \models \varphi$.*

Proof. We proceed by induction on φ. First, for p an atomic proposition, since morphisms preserves the valuation ℓ, we have $M', f(Y) \models p$ iff $M, Y \models p$. Thus the theorem is true for (possibly negated) atomic propositions. The case of the conjunction follows trivially from the induction hypothesis.

Suppose now that $M', f(X) \models K_a\varphi$. In order to show $M, X \models K_a\varphi$, assume that $a \in \chi(X \cap Y)$ for some facet Y, and let us prove $M, Y \models \varphi$. Let v be the a-colored vertex in $X \cap Y$. Then $f(v) \in f(X) \cap f(Y)$ and $\chi(f(v)) = a$. So $a \in \chi(f(X) \cap f(Y))$ and thus $M', f(Y) \models \varphi$. By induction hypothesis, we obtain $M, Y \models \varphi$. Finally, suppose that $M', f(X) \models C_A\varphi$. We want to show that $M, X \models C_A\varphi$, i.e., for every Y reachable from X following a sequence of simplexes sharing a A-colored vertex, $M, Y \models \varphi$. By the same reasoning as in the K_a case, $f(Y)$ is A-reachable from $f(X)$, so $M', f(Y) \models \varphi$, and thus $M, Y \models \varphi$. □

The restriction on φ forbids formulas saying something about what an agent does not know. Indeed, one can "gain" the knowledge that some agent does not know something; but this is not relevant information for solving the tasks that we have in mind.

3.2.2 Equivalence between simplicial and Kripke models

In this section, we prove the equivalence between Kripke models and simplicial models. In fact, this will require an additional assumption on Kripke models (namely, that they are *local*). But before we turn to models, we can first forget about the atomic propositions and compare Kripke frames and chromatic simplicial complexes. In this case, the equivalence between the two structures does not require any extra assumption.

Kripke frames and chromatic simplicial complexes

Let $\mathrm{Ag} = \{a_0, \dots, a_n\}$ be a set of $n+1$ agents. Recall that $\mathcal{K}_{\mathrm{Ag}}$ denotes the category of proper Kripke frames, with morphisms of Kripke frames as arrows. Let $\mathcal{S}_{\mathrm{Ag}}$ be the category of pure n-dimensional chromatic simplicial complexes colored by Ag, with chromatic simplicial maps for morphisms. The following theorem states that we can canonically associate a proper Kripke frame with a pure chromatic simplicial complex, and vice versa. In fact, this correspondence extends to morphisms, and thus we have an equivalence of categories, meaning that the two structures contain the same information.

Theorem 3.16. *$\mathcal{S}_{\mathrm{Ag}}$ and $\mathcal{K}_{\mathrm{Ag}}$ are equivalent categories.*

Proof. We construct functors $F : \mathcal{S}_{\mathrm{Ag}} \to \mathcal{K}_{\mathrm{Ag}}$ and $G : \mathcal{K}_{\mathrm{Ag}} \to \mathcal{S}_{\mathrm{Ag}}$ as follows.

Let $C = \langle V, S, \chi\rangle$ be a pure chromatic simplicial complex on the set of agents Ag. Its associated Kripke frame is $F(C) = \langle W, \sim\rangle$, where W is the set of facets of C, and the equivalence relation $\sim_a$, for each $a \in \mathrm{Ag}$, is generated by the relations $X \sim_a Y$ (for X and Y facets of C) if $a \in \chi(X \cap Y)$, that is, if X and Y share an a-colored vertex.

For a morphism $f : C \to D$ in $\mathcal{S}_{\mathrm{Ag}}$, we define $F(f) : F(C) \to F(D)$ which takes a facet X of C to its image $f(X)$, which is a facet of D since f is a chromatic map. Assume X and Y are facets of C such that $X \sim_a Y$ in $F(C)$, that is, $a \in \chi(X \cap Y)$. So there is a vertex $v \in V$ such that $v \in X \cap Y$ and $\chi(v) = a$. Then $f(v) \in f(X) \cap f(Y)$ and $\chi(f(v)) = a$, so $a \in \chi(f(X) \cap f(Y))$. Therefore, $f(X) \sim_a f(Y)$, and $F(f)$ is a morphism of Kripke frames.

Conversely, consider a Kripke frame $M = \langle W, \sim\rangle$ on the set of agents $\mathrm{Ag} = \{a_0, \dots, a_n\}$. Given an agent $a \in \mathrm{Ag}$ and a world $w \in W$, we write $[w]_a$ for the $\sim_a$-equivalence class of w, and $W/{\sim_a}$ for the set of all equivalence classes of $\sim_a$. We define $G(M) = \langle V, S, \chi\rangle$ as follows. Its set of vertices is defined as $V = \{(a, E) \mid a \in \mathrm{Ag}, E \in W/{\sim_a}\}$. A vertex $(a, E) \in V$ is colored by the first component, i.e., $\chi(a, E) = a$. For each world $w \in W$, we have one n-simplex $X_w = \{(a_0, [w]_{a_0}), \ldots, (a_n, [w]_{a_n})\}$. Then, the set S of simplices of $G(M)$ contains all the faces of these simplices. By definition, it is pure and n-dimensional. Notice that since the Kripke frame M is proper, all the simplices $(X_w)_{w \in W}$ are distinct, so there is a bijection between W and the facets of $G(M)$.

Now let $f : M \to N$ be a morphism in $\mathcal{K}_{\mathrm{Ag}}$. We define $G(f) : G(M) \to G(N)$ that maps a vertex $(a, [w]_a)$ of $G(M)$ to the vertex $(a, [f(w)]_a)$ of $G(N)$. It is easily checked that $G(f)$ is a chromatic simplicial map: a facet X_w of $G(M)$ is sent to $G(f)(X_w) = \{(a_0, [f(w)]_{a_0}), \ldots, (a_n, [f(w)]_{a_n})\}$, which is a facet of $G(N)$ by definition. Thus, a face of X_w is sent to a face of $G(f)(X_w)$.

Consider now a Kripke frame $M = \langle W, \sim \rangle$ in $\mathcal{K}_{\mathrm{Ag}}$ with agent set Ag. $FG(M)$ is the Kripke frame $N = \langle T, \sim' \rangle$ such that T is the set of facets of $G(M)$. But we have seen above that the facets X_w of $G(M)$ are in bijection with the worlds of W. Finally, in $FG(M)$, $X_w \sim'_{a_i} X_{w'}$ if and only if $a_i \in \chi(X_w \cap X_{w'})$, where χ is the coloring, in $G(M)$, of X_w and $X_{w'}$ which are facets of $G(M)$. But $a_i \in \chi(X_w \cap X_{w'})$ means that there is some vertex (a_i, E) which belongs both to X_w and $X_{w'}$. So, w and w' belong to the same equivalence class E, that is, $w \sim_{a_i} w'$. This proves that $FG(M)$ and M are isomorphic.

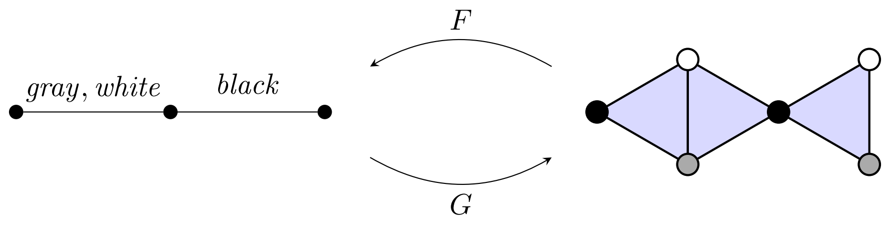
Conversely, let $C = \langle V, S, \chi \rangle \in \mathcal{S}_{\mathrm{Ag}}$ be a pure chromatic simplicial complex. Given a vertex $v \in V$ colored by a_i, we denote by $\widehat{v}$ the set of all the facets of C which contain v. By definition, $\widehat{v}$ is an equivalence class of the relation $\sim_{a_i}$ in the Kripke frame $F(C)$. Therefore, the pair $f(v) := (a_i, \widehat{v})$ is a vertex of $GF(C)$. It is easy to check that the map f is a bijection between the vertices of C and those of $GF(C)$. Moreover, the facets of $GF(C)$ are of the form X_w, where w is a world of $F(C)$, that is, a facet of C. So, we have a bijection between the facets of C and those of $GF(C)$. This bijection is given precisely by f. Indeed, a facet $Y = \{v_0, \ldots, v_n\}$ of C is sent to $f(Y) = \{(a_0, \widehat{v_0}), \ldots, (a_n, \widehat{v_n})\}$. But since each vertex $v_i \in Y$, we have $\widehat{v_i} = [Y]_{a_i}$, and therefore $f(Y) = X_Y$. Hence C and $GF(C)$ are isomorphic and therefore, $\mathcal{S}_{\mathrm{Ag}}$ and $\mathcal{K}_{\mathrm{Ag}}$ are equivalent categories. □

Remark 3.17. A more categorical definition of the simplicial complex $G(M)$ associated to a Kripke frame M is the following. Let Δ^{Ag} be the pure n-dimensional chromatic simplicial complex consisting of just one facet, colored by Ag. We define $G(M)$ as a quotient of the coproduct of W-many copies of Δ^{Ag},

$$G(M) := \left(\coprod_{w \in W} \Delta^{\mathrm{Ag}} \right) / R$$

where the equivalence relation R is defined by $v_i^w \, R \, v_i^{w'}$ iff $w \sim_{a_i} w'$, where v_i^w denotes the a_i-colored vertex of the w-th copy of Δ^{Ag}. Intuitively, what this means is that we take one n-simplex for each $w \in W$, and glue them together according to the indistiguishability relation. Taking a quotient in the category of simplicial complexes simply amounts to taking equivalence classes as vertices, like we did in the proof of Theorem 3.16.

Example 3.18. The picture below shows a Kripke frame (left) and its associated chromatic simplicial complex (right). The three agents, named black, gray, white, are represented as colors on the vertices of the simplicial complex. The three worlds of the Kripke frame correspond to the three triangles (i.e., 2-dimensional simplexes) of the simplicial complex. The two worlds indistinguishable by agent black, are glued along their black vertex; the two worlds indistinguishable by agents gray and white are glued along the gray-and-white edge.



Kripke models and simplicial models

Local Kripke models. As in the case of simplicial models, we fix the set of atomic propositions $\mathrm{At} = \{p_{a,x} \mid a \in \mathrm{Ag}, x \in \mathcal{V}\}$, and for a given agent a, $\mathrm{At}_a = \{p_{a,x} \mid x \in \mathcal{V}\}$. A Kripke model $M = \langle W, \sim, L\rangle$ is *local* if for every agent $a \in \mathrm{Ag}$, $w \sim_a w'$ implies $L(w) \cap \mathrm{At}_a = L(w') \cap \mathrm{At}_a$, i.e., an agent always knows its own values. We write $\mathcal{KM}^{\mathrm{loc}}_{\mathrm{Ag},\mathrm{At}}$ for the category of local proper Kripke models.

Remark 3.19. It is unusual in the literature to consider classes of Kripke models with this kind of semantic restriction. Most of the time, the restrictions are on the Kripke frame, and once a class of frames is chosen, any labeling of atomic propositions is allowed. One notable occurrence of this is the notion of *hypercube system* [89], which is closely related to our locality assumption. Hypercube systems are special cases of *interpreted systems* [106], which are used to model knowledge in distributed computing.

We can now extend the two maps F and G of Theorem 3.16 to an equivalence between simplicial models and local proper Kripke models.

Theorem 3.20. *$\mathcal{SM}_{\mathrm{Ag},\mathrm{At}}$ and $\mathcal{KM}^{loc}_{\mathrm{Ag},\mathrm{At}}$ are equivalent categories.*

Proof. We describe the functors $F : \mathcal{SM} \to \mathcal{KM}^{\mathrm{loc}}$ and $G : \mathcal{KM}^{\mathrm{loc}} \to \mathcal{SM}$. On the underlying Kripke frame and simplicial complex, they act the same as in the proof of Theorem 3.16.

Given a simplicial model $M = \langle V, S, \chi, \ell\rangle$, we associate the Kripke model $F(M) = \langle \mathcal{F}(M), \sim, L\rangle$ where the labeling L of a facet $X \in \mathcal{F}(M)$ is given by $L(X) = \bigcup_{v\in X} \ell(v)$. This Kripke model is local since $X \sim_a Y$ means that X and Y share an a-colored vertex v, so $L(X) \cap \mathrm{At}_a = L(Y) \cap \mathrm{At}_a = \ell(v)$.

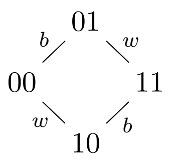
Conversely, given a Kripke model $M = \langle W, \sim, L\rangle$, the underlying simplicial complex of $G(M)$ has vertices of the form $(a, [w]_a)$ where a is an agent and $[w]_a$ is the equivalence class of w for the relation $\sim_a$. We label this vertex (which is colored by a) by $\ell(a, [w]_a) = L(w) \cap \mathrm{At}_a$. This is well defined because two vertices $(a, [w]_a)$ and $(a, [w']_a)$ are identified whenever $w \sim_a w'$. Since M is local, we have $L(w) \cap \mathrm{At}_a = L(w') \cap \mathrm{At}_a$, and the labeling ℓ does not depend on the choice of w.

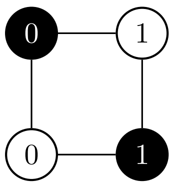
The action of F and G on morphisms is the same as in Theorem 3.16. It is easy to check that the additional properties of morphisms between models are verified. Checking that $FG(M) \simeq M$ and $GF(M) \simeq M$ also works the same as in the previous theorem. □

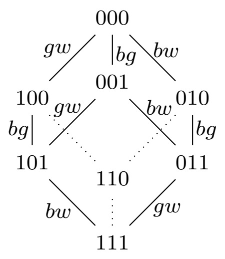
Example 3.21. The figure below shows the binary input complex and its associated Kripke model, for 2 and 3 agents. Each agent is given a binary value 0 or 1, but does not know which value has been given to other agents. So, every possible combination of 0's and 1's is a possible world.

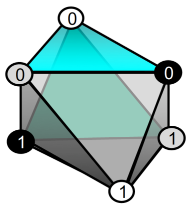
In the Kripke model, the agents are called b, g, w (short for *black*, *gray* and *white*), and the labeling L of the possible worlds is represented as a sequence of values, e.g., 101, representing the values given to the agents b, g, w (in that order). In the 3-agents case, the labels of the dotted edges have been omitted to avoid overloading the picture, as well as other edges that can be deduced by transitivity.

In the simplicial model, agents are represented as colors (black, gray, and white). The labeling ℓ is represented as a single value in a vertex, e.g., "1" in a gray vertex means that agent g has value 1. The possible worlds correspond to edges in the 2-agents case, and triangles in the 3-agents case.









It is well known in the context of distributed computing [64] that the binary input simplicial complex for $n+1$ agents is a n-dimensional sphere. In epistemic logic, this is an example of a hypercube system [89].

We can show that our simplicial definition of truth (Definition 3.12) agrees with the usual one (Definition 3.3) on the corresponding Kripke model. To distinguish the two, we write them $\models_{\mathcal{S}}$ and $\models_{\mathcal{K}}$ in the next proposition.

Proposition 3.22. *Given a simplicial model M and a facet X, $M, X \models_{\mathcal{S}} \varphi$ iff $F(M), X \models_{\mathcal{K}} \varphi$. Conversely, given a local proper Kripke model N and world w, $N, w \models_{\mathcal{K}} \varphi$ iff $G(N), X_w \models_{\mathcal{S}} \varphi$, where X_w is the facet of $G(N)$ defined in the proof of Theorem 3.16.*

Proof. The first claim is proved by a straightforward induction on the formula φ. The second claim reduces to the first one by applying Theorem 3.20, since $N \simeq FG(N)$. □

3.2.3 Soundness and completeness

It is well-known that the axiom system $\mathbf{S5_n}$ is sound and complete for $\mathcal{L}_K$ with respect to the class of Kripke models [27]. Since we restrict here to local Kripke models, we need to add the following axiom (or axiom schema, if $\mathcal{V}$ is infinite), saying that every agent knows which values it holds:

$$\mathbf{Loc} \;=\; \bigwedge_{a \in A, x \in \mathcal{V}} K_a(p_{a,x}) \vee K_a(\neg p_{a,x})$$

Remark 3.23. Note that a similar axiomatization has been considered in [89] to study hypercube systems, also axiomatized by a formula depending on atoms (and not just on the underlying frame structure). In their case, the axiom that they introduce is even stronger than ours, saying that an agent knows its own values, but also that it knows *only* its value. These formulas depending on atoms are rather rare in the literature, but can also be found in distributed computing related epistemic logics, where for example public announcements can be lost during transmission [12].

Proposition 3.24. *The axiom system $\mathbf{S5_n} + \mathbf{Loc}$ is sound and complete for the language $\mathcal{L}_K$ w.r.t. the class of local proper Kripke models.*

Proof. We adapt the proof of [27] for $\mathbf{S5_n}$, which works for non-necessarily local Kripke models.
Soundness: If φ is provable in $\mathbf{S5_n} + \mathbf{Loc}$, then it is true in every world of every proper local Kripke model. It is well known that all the deduction rules of $\mathbf{S5_n}$ are admissible in all Kripke models; we only have to check that *local* Kripke models moreover satisfy the $\mathbf{Loc}$ axiom, which is straightforward.

Completeness: If φ is true in every proper local Kripke model, then it is provable. The usual proof for $\mathbf{S5_n}$ [27] proceeds by contraposition: assuming φ is not provable, we construct a model in which φ is false. This model is called the *canonical model*.

A set Γ of formulas is *consistent* if $\Gamma \not\vdash_{\mathbf{S5_n}+\mathbf{Loc}} \bot$, and it is *maximal* if there is no greater consistent set $\Gamma' \supsetneq \Gamma$. The canonical model M^c is defined as $M^c = \langle S^c, \sim^c, L^c \rangle$, where:

- $S^c = \{\Gamma \mid \Gamma$ is a consistent and maximal set of formulas$\}$
- $\Gamma \sim_a^c \Delta$ iff $\{K_a\varphi \mid K_a\varphi \in \Gamma\} = \{K_a\varphi \mid K_a\varphi \in \Delta\}$
- $L^c(\Gamma) = \Gamma \cap \mathrm{At}$

The usual machinery on the canonical model shows that if φ is not provable in $\mathbf{S5_n} + \mathbf{Loc}$, then there is Γ such that $M^c, \Gamma \not\models \varphi$. We only have to check that M^c is proper and local. First notice that if Γ is consistent and maximal, then $p_{a,x} \in \Gamma \Leftrightarrow K_a(p_{a,x}) \in \Gamma$. The right-to-left direction follows from the truth axiom $K_a\varphi \Rightarrow \varphi$. For the converse, assume $p_{a,x} \in \Gamma$ and $K_a(p_{a,x}) \notin \Gamma$. Then we must have $K_a(\neg p_{a,x}) \in \Gamma$ because of the $\mathbf{Loc}$ axiom, which implies $\neg p_{a,x} \in \Gamma$, and thus Γ would be inconsistent.

M^c is local: assume $\Gamma \sim_a^c \Delta$, we show that $L^c(\Gamma) \cap \mathrm{At}_a = L^c(\Delta) \cap \mathrm{At}_a$. Indeed, $p_{a,x} \in \Gamma \Leftrightarrow K_a(p_{a,x}) \in \Gamma \Leftrightarrow K_a(p_{a,x}) \in \Delta \Leftrightarrow p_{a,x} \in \Delta$.

M^c is proper: assume that for all a, $\Gamma \sim_a^c \Delta$. Then we show by a straightforward induction on φ that $\varphi \in \Gamma \Leftrightarrow \varphi \in \Delta$, i.e., $\Gamma = \Delta$. □

Remark 3.25. Notice that we could have omitted the "proper" assumption: $\mathbf{S5_n} + \mathbf{Loc}$ is also sound and complete w.r.t. the class of local Kripke models. The proof is the same as for Proposition 3.24, without the last line.

Corollary 3.26. *The axiom system* $\mathbf{S5_n} + \mathbf{Loc}$ *is sound and complete w.r.t. the class of simplicial models.*

Proof. Using Proposition 3.22, we can now transpose Proposition 3.24 to simplicial models. Suppose that a formula φ is true for every local proper Kripke model and any world. Then given a simplicial model M and facet X, since by assumption $F(M), X \models_{\mathcal{K}} \varphi$, we also have $M, X \models_{\mathcal{S}} \varphi$ by Proposition 3.22. So φ is true in every simplicial model. Similarly, the converse also holds. □

Remark 3.27. When we include common knowledge in our formulas, the proof of soundness and completeness of $\mathbf{S5C_n}$ w.r.t. the class of Kripke models [27] is very similar to the one for $\mathbf{S5_n}$. It seems that it can be adapted to account for the locality assumption and the $\mathbf{Loc}$ axiom, but there are some technical details in the proof of completeness that we have not worked out completely yet.

The proof of soundness can be adapted straightforwardly, since the rules of $\mathbf{S5C_n}$ are known to be admissible in all Kripke models, and moreover the $\mathbf{Loc}$ axiom is admissible in the local Kripke models. For completeness, due to some technical difficulties, it is not possible to use *infinite* maximal consistent sets of formulas, because the system $\mathbf{S5C_n}$ is not compact. So, instead, given the formula φ that we are interested in, we construct a model based on maximal consistent sets for a finite fragment $\mathsf{cl}(\varphi)$ of the language $\mathcal{L}_{CK}$, called the *closure* of φ. Then, the canonical model for $\mathsf{cl}(\varphi)$ is defined as in the proof of Proposition 3.24. As before, all we have to show is that this canonical model is local (and proper), but in fact this might not be the case. One way to fix it seems to be to extend the definition of $\mathsf{cl}(\varphi)$, to say that if an atom $p_{a,x} \in \mathsf{cl}(\varphi)$, then $K_a\, p_{a,x} \in \mathsf{cl}(\varphi)$, and similarly for negated atoms.

3.2.4 Relaxing the locality condition

Let us assume we are given a Kripke model, and we want to understand it geometrically. Theorem 3.20 gives us a way to translate a Kripke model into a simplicial model, with two restrictions: the original model must be *proper* and *local*. Being proper is not a very stong requirement, since many practical situations usually produce proper Kripke frames. However, the locality condition might seem quite restrictive. Indeed, we can think of many situations where some properties of the worlds do not take the form of a value held by one of the agents.

For instance, in Example 3.14, we could have wanted an atomic proposition representing the value of the remaining card that was not dealt to any of the agents. We cannot assign the hidden card to one of the agents, because none of them knows what its value is. Thus, this is a non-local situation. However, in this particular example, we did not need an atomic proposition for the hidden card, because this information would be redundant: we can recover it from the values of the cards that were dealt to the agents.

In this section, we will show that it is always possible to do this trick. Starting with a (proper) non-local Kripke model M, we will construct a local Kripke model M^{loc} (with a new set of atomic propositions), such that every formula φ on M can be translated to an equivalent formula φ^{loc} on M^{loc}.

In this section only, At will denote an arbitrary set of atomic propositions. Let $M = \langle W, \sim, L\rangle$ be a proper Kripke model on At. For each agent $a \in \text{Ag}$, we write $X_a = W/\!\sim_a$ for the set of equivalence classes of $\sim_a$. For $w \in W$, we write $[w]_a \in X_a$ the equivalence class of w. Then we take one atomic proposition symbol for each agent and equivalence class, yielding a new set of atomic propositions At^{loc}:

$$\text{At}^{\text{loc}} = \{E_{a,x} \mid a \in \text{Ag and } x \in X_a\}$$

We now define a Kripke model $M^{\text{loc}} = \langle W, \sim, L^{\text{loc}}\rangle$ on At^{loc}, with the same underlying Kripke frame $\langle W, \sim\rangle$ as M, and the labeling $L^{\text{loc}} : W \to \mathscr{P}(\text{At}^{\text{loc}})$ is defined as $L^{\text{loc}}(w) = \{E_{a,[w]_a} \mid a \in \text{Ag}\}$.

Proposition 3.28. *The Kripke model M^{loc} is proper and local.*

Proof. It is proper since it has the same underlying Kripke frame as M, which is proper. To check locality, suppose $w \sim_a w'$. Then $L^{\text{loc}}(w) \cap \text{At}_a^{\text{loc}} = \{E_{a,[w]_a}\} = L^{\text{loc}}(w') \cap \text{At}_a^{\text{loc}}$, since $[w]_a = [w']_a$. □

We now want to translate every formula $\varphi \in \mathcal{L}_{CK}(\text{At})$ to a formula $\varphi^{\text{loc}} \in \mathcal{L}_{CK}(\text{At}^{\text{loc}})$, such that for every world $w \in W$, $M, w \models \varphi$ iff $M^{\text{loc}}, w \models \varphi^{\text{loc}}$. First, remark that, since M is proper, every world $w \in W$ is uniquely determined by the set of equivalence relations $\{[w]_a \mid a \in \text{Ag}\}$. Thus, we can translate every atomic proposition $p \in \text{At}$ as follows. Let $V_p = \{w \in W \mid p \in L(w)\}$ be the set of worlds where p is true in M. We then define $p^{\text{loc}} = \bigvee_{w \in V_p} \bigwedge_{a \in \text{Ag}} E_{a,[w]_a}$. The above remark says that, in M^{loc}, the formula $\bigwedge_{a \in \text{Ag}} E_{a,[w]_a}$ is true exactly in the world w. So, the formula p^{loc} is true exactly in the worlds of V_p, i.e., $M, w \models p$ iff $M^{\text{loc}}, w \models p^{\text{loc}}$. It is now straightforward to extend this translation to every formula $\varphi \in \mathcal{L}_{CK}(\text{At})$: we just replace every occurrence of an atomic proposition p by p^{loc}. This gives us a formula $\varphi^{\text{loc}} \in \mathcal{L}_{CK}(\text{At}^{\text{loc}})$, and by a trivial induction we get:

Theorem 3.29. *For every $w \in W$, $M, w \models \varphi$ iff $M^{loc}, w \models \varphi^{loc}$.*

Therefore, through this translation, even the non-local Kripke models can be interpreted geometrically. The only condition which is really relevant is that the underlying Kripke frame must be proper.

3.2.5 A simplicial product update model

Our notion of simplicial model allows us to interpret formulas of epistemic logic $\mathcal{L}_{CK}$. In order to interpret *dynamic* epistemic logic, we need to define the product-update operation on simplicial models. Of course, one way to do it would be, given a simplicial model M, to transform it into a Kripke model $F(M)$ using the equivalence of Section 3.2.2, apply the usual product-update construction to get $F(M)[\mathcal{A}]$, and then transform the result into a simplicial model again, which gives $G(F(M)[\mathcal{A}])$. This last step is possible only if the product-update model $F(M)[\mathcal{A}]$ is local and proper, which is the case as soon as $\mathcal{A}$ is proper:

Proposition 3.30. *Let M be a local proper Kripke model and $\mathcal{A} = \langle T, \sim, \mathsf{pre}\rangle$ a proper action model, then $M[\mathcal{A}]$ is proper and local.*

Proof. $M[\mathcal{A}]$ is proper: let (w,t) and (w',t') be two distinct worlds of $M[\mathcal{A}]$. Then either $w \neq w'$ or $t \neq t'$, and in both cases, since M and $\mathcal{A}$ are proper, at least one agent can distinguish between the two. Now, $M[\mathcal{A}]$ is local: suppose $(w,t) \sim_a^{[\mathcal{A}]} (w',t')$. Then in particular $w \sim_a w'$ and since M is local, $L(w) \cap \mathrm{At}_a = L(w') \cap \mathrm{At}_a$. The same goes for $L[\mathcal{A}]$ since it just copies L. □

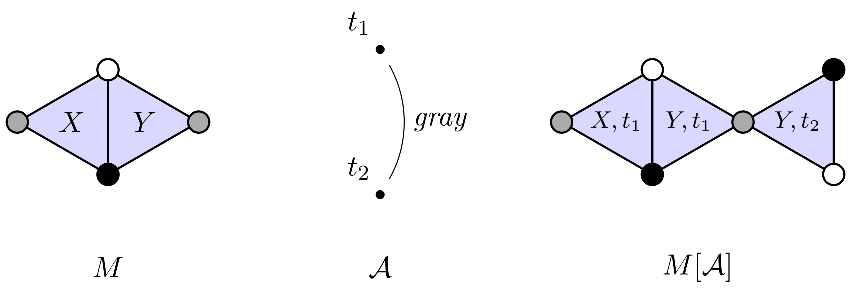
Instead of using $G(F(M)[\mathcal{A}])$, which is rather cumbersome, we give an explicit construction of a simplicial model $M[\mathcal{A}]$, where M is a simplicial model and $\mathcal{A}$ and action model. We call this the "hybrid" product-update operation. We also describe a fully simplicial version, where the action model $\mathcal{A}$ is also given as a simplicial complex. Finally, we prove that these three versions of the product-update model yield the same result.

The hybrid product-update. We now define a variant of the product-update, where we start with a simplicial model M and a usual action model $\mathcal{A}$; and the product-update model $M[\mathcal{A}]$ that we obtain at the end should also be a simplicial model. To understand the following definitions, the reader should keep in mind the translation between Kripke models and simplicial models (see Theorem 3.20).

Intuitively, the facets of $M[\mathcal{A}]$ should correspond to pairs (X,t) where $X \in \mathcal{F}(M)$ is a world of M and $t \in T$ is an action of $\mathcal{A}$, such that $M, X \models \mathsf{pre}(t)$. Moreover, two such facets (X,t) and (Y,t') should be glued along their a-colored vertex whenever $a \in \chi(X \cap Y)$ and $t \sim_a t'$.

Definition 3.31 (Hybrid product-update)**.** Let $M = \langle V, S, \chi, \ell\rangle$ be a simplicial model, and $\mathcal{A} = \langle T, \sim, \mathsf{pre}\rangle$ be a proper action model. The *hybrid product-update model* $M[\mathcal{A}] = \langle V[\mathcal{A}], S[\mathcal{A}], \chi[\mathcal{A}], \ell[\mathcal{A}]\rangle$ is defined as follows. The vertices $V[\mathcal{A}]$ are pairs (v, E) where $v \in V$ is a vertex of M; E is an equivalence class of $\sim_{\chi(v)}$; and v belongs to some facet $X \in \mathcal{F}(M)$ such that there exists $t \in E$ such that $M, X \models \mathsf{pre}(t)$. Such a vertex keeps the color and labeling of its first component: $\chi[\mathcal{A}](v,E) = \chi(v)$ and $\ell[\mathcal{A}](v,E) = \ell(v)$. For each pair (X,t) such that $M, X \models \mathsf{pre}(t)$, where $X = \{v_0, \ldots, v_n\}$ is a facet of M and $t \in T$, we get a facet of $M[\mathcal{A}]$ defined as $\{(v_0, [t]_{\chi(v_0)}), \ldots, (v_n, [t]_{\chi(v_n)})\}$. The set of simplices $S[\mathcal{A}]$ is the set of all their faces.

Example 3.32. Below is depicted an example with three agents, where M consists of two worlds X and Y, and $\mathcal{A}$ has two actions t_1 and t_2. The gray agent is the only one which cannot distinguish between the two actions. The precondition $\mathsf{pre}(t_1)$ is true in both X and Y, but $\mathsf{pre}(t_2)$ is true only in Y.



The advantage of this "hybrid" product update is that it mimics closely the classic definition; thus, it should be easier to understand for readers already familiar with DEL. In the following, we define yet another product update operation, where both M and $\mathcal{A}$ are simplicial. This will help us uncover the geometric structure of this operation.

The fully simplicial product-update. We can extend the translation of DEL into simplicial complexes by noticing that the action model $\mathcal{A} = \langle T, \sim, \mathsf{pre} \rangle$ merely consists of a Kripke frame $\langle T, \sim \rangle$ along with precondition formulas. By applying Theorem 3.16 to the underlying Kripke frame, we obtain a simplicial counterpart of action models.

Definition 3.33 (Simplicial action model)**.** A *simplicial action model* $\langle V_T, S_T, \chi_T, \mathsf{pre} \rangle$ consists of a pure chromatic simplicial complex $T = \langle V_T, S_T, \chi_T \rangle$, where the facets $\mathcal{F}(T)$ represent communicative *actions*, and $\mathsf{pre} : \mathcal{F}(T) \to \mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ assigns to each facet $X \in \mathcal{F}(T)$ a precondition formula in $\mathcal{L}_{CK}$.

Before defining the corresponding product-update operation, we first need to define the cartesian products in the category of chromatic simplicial complexes.

Definition 3.34 (Cartesian product in $\mathcal{S}_{\mathrm{Ag}}$)**.** Given two pure chromatic simplicial complexes of dimension n, $C = \langle V_C, S_C, \chi_C \rangle$ and $T = \langle V_T, S_T, \chi_T \rangle$, the cartesian product $C \times T$ is the following pure chromatic simplicial complex of dimension n. Its vertices are of the form (u, v) with $u \in V_C$ and $v \in V_T$ such that $\chi_C(u) = \chi_T(v)$; the color of (u, v) is $\chi(u, v) := \chi_C(u) = \chi_T(v)$. Its simplexes are of the form $X \times Y = \{(u_0, v_0), \dots, (u_k, v_k)\}$ where $X = \{u_0, \dots, u_k\} \in C$, $Y = \{v_0, \dots, v_k\} \in T$, and $\chi(u_i) = \chi(v_i)$.

Definition 3.35 (Simplicial product-update)**.** Given a simplicial model $M = \langle V, S, \chi, \ell \rangle$, and a simplicial action model $\mathcal{A} = \langle V_T, S_T, \chi_T, \mathsf{pre} \rangle$, we define the *simplicial product-update model* $M[\mathcal{A}] = \langle V[\mathcal{A}], S[\mathcal{A}], \chi[\mathcal{A}], \ell[\mathcal{A}] \rangle$ as a simplicial model whose underlying simplicial complex is a sub-complex of the cartesian product $C \times T$, induced by all the facets of the form $X \times Y$ such that $M, X \models \mathsf{pre}(Y)$. The valuation $\ell : V[\mathcal{A}] \to \mathscr{P}(\mathrm{At})$ at a pair (u, v) is just as it was at u, i.e., $\ell[\mathcal{A}](u, v) := \ell(u)$.

Recall from Theorem 3.20 the two functors F and G that define an equivalence of categories between simplicial models and Kripke models. We have a similar correspondence between action models and simplicial action models, which we still write F and G. On the underlying Kripke frame and simplicial complex they are the same as before; and the precondition of an action point is just copied to the corresponding facet. Proposition 3.36 says that the "hybrid" product update and the "fully simplicial" one give the same result.

Proposition 3.36. *Let M be a simplicial model, and $\mathcal{A}$ a proper action model. $G(\mathcal{A})$ is the associated simplicial action model. Then the simplicial models $M[\mathcal{A}]$ and $M[G(\mathcal{A})]$ are isomorphic.*

Proof. By definition of the hybrid product update $M[\mathcal{A}]$, its vertices are of the form (u, E) where u is a vertex of M and E an equivalence class of $\sim_{\chi(u)}$. On the other hand, the vertices of the product update $M[G(\mathcal{A})]$ are of the form (u, v) where u is a vertex of M and v a vertex of $G(\mathcal{A})$. But by definition of G (see the proof of Theorem 3.16), v corresponds precisely to an equivalence class of $\sim_{\chi(u)}$. This gives us a bijection between the vertices of $M[\mathcal{A}]$ and those of $M[G(\mathcal{A})]$. There remains to check that both directions of the bijection are actually simplicial maps, which is straightforward. $\square$

The next proposition says that the usual product update operation agrees with the simplicial one:

Proposition 3.37. *Consider a simplicial model M and simplicial action model $\mathcal{A}$, and their corresponding Kripke model $F(M)$ and action model $F(\mathcal{A})$. Then, the Kripke models $F(M[\mathcal{A}])$ and $F(M)[F(\mathcal{A})]$ are isomorphic. The same is true for G, starting with a Kripke model M and action model $\mathcal{A}$.*

Proof. The main observation is that both constructions of product update model rely on a notion of cartesian product (in the category of pure chromatic simplicial complexes for $M[\mathcal{A}]$, and in the category of Kripke frames for $F(M)[F(\mathcal{A})]$). These are both cartesian products in the categorical sense, therefore they are preserved by the functor F because it is part of an equivalence of category: given C and T the underlying chromatic simplicial complexes of M and $\mathcal{A}$ respectively, we have $F(C \times T) \simeq F(C) \times F(T)$. Intuitively, a world of $F(C \times T)$ is a facet of $C \times T$ of the form $X \times Y$, which is entirely determined by the two facets $X \in F(C)$ and $Y \in F(T)$.

Then, $M[\mathcal{A}]$ is defined as the sub-complex of $C \times T$ consisting of all facets $X \times Y$ such that $\mathsf{pre}(Y)$ holds in X, that is, $M, X \models_{\mathcal{S}} \mathsf{pre}(Y)$. On the other hand, $F(M)[F(\mathcal{A})]$ is defined as the sub-frame of $F(C) \times F(T)$ consisting of all worlds (X, Y) such that $\mathsf{pre}(Y)$ holds in X, that is, $F(M), X \models_{\mathcal{K}} \mathsf{pre}(Y)$. By Proposition 3.22, these two conditions are equivalent, so the underlying Kripke frames of $F(M[\mathcal{A}])$ and $F(M)[F(\mathcal{A})]$ are isomorphic. Moreover, in both cases, the labeling L of atomic propositions is just copied from the first component, so they are also isomorphic as Kripke models. $\square$

As a direct consequence of Propositions 3.37 and 3.36, we get the equivalence between the hybrid product update $M[\mathcal{A}]$ and the construction $G(F(M)[\mathcal{A}])$ mentioned in the beginning of the section:

Proposition 3.38. *Given a simplicial model M and an action model $\mathcal{A}$, we have $G(F(M)[\mathcal{A}]) \simeq M[\mathcal{A}]$.*

Proof. By the second part of Proposition 3.37, applied using the Kripke model $F(M)$ and the action model $\mathcal{A}$, we get $G(F(M)[\mathcal{A}]) \simeq GF(M)[G(\mathcal{A})]$. Using the fact that $GF(M) \simeq M$, and Proposition 3.36, we obtain $GF(M)[G(\mathcal{A})] \simeq M[\mathcal{A}]$. $\square$

3.3 Distributed computing through Dynamic Epistemic Logic

We now use the simplicial models of DEL defined in Section 3.2 to model distributed computability. It is clear that simplicial models are closely related to the labeled chromatic simplicial complexes of Chapter 1. For instance, the simplicial model of Example 3.21 corresponds to the *input complex* of the binary consensus task (Example 1.20). What is less obvious is the relationship between the *carrier* *maps* of Chapter 1, and the product-update construction: both of them model what happens when the agents/processes communicate and learn new information about the world.

From now on, the set Ag of agents will correspond to the $n+1$ *processes* in a distributed system. Consider a simplicial model $\mathcal{I}$ representing the possible initial states of the system, for instance, the binary input simplicial model of Example 3.21. We will model a protocol as an action model $\mathcal{A}$, so that the product update model $\mathcal{I}[\mathcal{A}]$ represents the knowledge of the agents after executing the protocol. The simplicial model $\mathcal{I}[\mathcal{A}]$ corresponds to the *protocol complex* of distributed computability.

Similarly, action models can be used to specify a concurrent *task*, but the correspondence with the approach of Chapter 1 is not exactly the same. A task is specified by an action model $\mathcal{T}$, which describes the output values that the agents should be able to produce, as well as preconditions specifying which inputs are allowed to produce which outputs. Here, the action model $\mathcal{T}$ itself will correspond to the *output complex* of distributed computing. The product update of the input model $\mathcal{I}$ with $\mathcal{T}$ yields an epistemic model $\mathcal{I}[\mathcal{T}]$, which is not usually considered in distributed computing. A possible intuitive explanation of this model is that it represents the amount of knowledge that the agents should be able to acquire in order to solve the task. Once the protocol action model $\mathcal{A}$ and the task action model $\mathcal{T}$ are specified, we can define what it means for a protocol to *solve* a task, i.e., give an analogue of Definition 1.29. The protocol $\mathcal{A}$ solves the task if there exists a morphism δ that makes the diagram on the right commute. On the diagram, the two simplicial maps labeled π are simply projections on the first component (recall that the product update models $\mathcal{I}[\mathcal{A}]$ and $\mathcal{I}[\mathcal{T}]$ are subcomplexes of a cartesian product). Notice how this diagram differs from the one of Definition 1.29: the two carrier maps have been replaced by simplicial maps going in the other direction.

$$\begin{tikzcd}
 & \mathcal{I}[\mathcal{A}] \arrow[dl, "\pi"'] \arrow[d, dashed, "\delta"] \\
\mathcal{I} & \mathcal{I}[\mathcal{T}] \arrow[l, "\pi"]
\end{tikzcd}$$

3.3.1 Protocols as action models

We focus on the specific setting of asynchronous wait-free processes communicating using shared read/write memory. In fact, we present two computational models which are both known to be equivalent to this setting in terms of computational power. First, we define the *layered message-passing model* for two processes, which is very easy to describe. To be able to work with any number of processes, we then define the *immediate snapshot model* (as in Example 1.26).

The layered message-passing action model

We start with an informal explanation of the *layered message-passing model* for two processes. More details about this model can be found in [64].

Let the processes be b, w, in order to draw them in the pictures with colors black and white. In the *layered message-passing* model, computation is synchronous: b and w take steps at the same time. We will call each such step a *layer*. In each layer, b and w both send a message to each other. Moreover, communication in this model is *unreliable*: in each layer, at most one message may fail to arrive. So, either one or two messages will be received. This is a *full information* model, in the sense that each time a process sends a message, the message consists of its local state (i.e., all the information currently known to the process), and each time it receives a message, it appends it to its own local state (remembers everything). A protocol is defined by the number N of layers that the processes execute. Then, each process should produce an output value based on its state at the end of the last layer. A decision function δ specifies the output value of each process at the end of the last layer.

Given an initial configuration, an execution can be specified by a sequence of N symbols over the alphabet $\{\perp, b, w\}$, where the i-th symbol is the name of the process whose message was lost in the i-th layer. More precisely, if the i-th symbol is $\perp$ then in the i-th layer both messages arrived, and if the i-th symbol is b (resp. w) then only b's message failed to arrive (resp. w) in the i-th layer. As an example, $\perp bw$ corresponds to an execution in which both processes have received each other's message at layer one, then b received the message from w but w did not receive the message from b at layer two, and finally at layer three, w received the message from b but b did not receive the message from w.

Notice that there are three 1-layer executions, namely $\perp$, b and w, but from the point of view of process b, there are only two distinguished cases: (i) either it did not receive a message, in which case it knows for sure that the execution that occurred was w, or (ii) it did receive a message from w, in which case the execution could have been either b or $\perp$. Thus, for the black process, the executions b and $\perp$ are indistinguishable.

As an action model. Consider the situation where the agents $\mathrm{Ag} = \{b, w\}$ start in an initial configuration, defined by input values given to each agent. The values are local, in the sense that each agent knows its own initial value, but not necessarily the values given to other agents. The agents communicate with each other via the layered message-passing model described above.

Let V^{in} be an arbitrary domain of *input values*, and take the following set of atomic propositions $\mathrm{At} = \{\mathsf{input}_a^x \mid a \in \mathrm{Ag}, x \in V^{in}\}$. Consider a simplicial model $\mathcal{I} = \langle V_\mathcal{I}, S_\mathcal{I}, \chi, \ell\rangle$ called the *input simplicial model*. Moreover, we assume that for each vertex $v \in V_\mathcal{I}$, corresponding to some agent $a = \chi(v)$, the labeling $\ell(v) \subseteq \mathrm{At}_a$ is a singleton, assigning to the agent a its private input value. A facet $X \in \mathcal{F}(\mathcal{I})$ represents a possible initial configuration, where each agent has been given an input value.

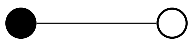
Definition 3.39. The action model $\mathcal{MP}_N = \langle T, \sim, \mathsf{pre}\rangle$ corresponding to N layers is defined as follows. Let L_N be the set of all sequences of N symbols over the alphabet $\{\perp, b, w\}$. Then, we take $T = L_N \times \mathcal{F}(\mathcal{I})$. An action (α, X), where $\alpha \in L_N$ and $X \in \mathcal{F}(\mathcal{I})$ represents a possible execution starting in the initial configuration X. We write X_a for the input value assigned to agent a in the input simplex X. Then, $\mathsf{pre} : T \to \mathcal{L}_{CK}$ assigns to each $(\alpha, X) \in T$ a precondition formula $\mathsf{pre}(\alpha, X)$ which holds exactly in X (formally, we take $\mathsf{pre}(\alpha, X) = \bigwedge_{a\in\mathrm{Ag}} \mathsf{input}_a^{X_a}$). To define the indistinguishability relation $\sim_a$, we proceed by induction on N. For $N = 0$, we define $(\varnothing, X) \sim_a (\varnothing, Y)$ when $X_a = Y_a$, since process a only sees its own local state. Once the indistinguishability relations of $\mathcal{MP}_N$ have been defined, we define $\sim_a$ on $\mathcal{MP}_{N+1}$ as follows. Let $\alpha, \beta \in L_N$ and $p, q \in \{\perp, b, w\}$. We define $(\alpha \cdot p, X) \sim_b (\beta \cdot q, Y)$ if either:

(i) $p = q = w$ and $(\alpha, X) \sim_b (\beta, Y)$, or

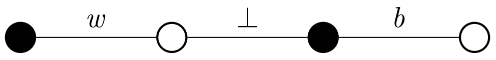
(ii) $p, q \in \{\perp, b\}$ and $X = Y$ and $\alpha = \beta$,

and similarly for $\sim_w$, with the role of b and w reversed.

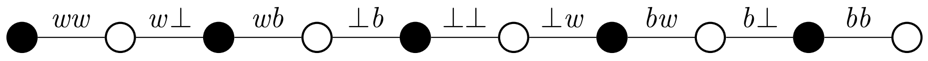
Intuitively, either (i) no message was received, and the uncertainty from the previous layers remain; or (ii) a message was received, and the process b can see the whole history, except that it does not know whether the last layer was b or $\perp$. To see what the effect of this action model is, let us start with an input model $\mathcal{I}$ with only one input configuration X (input values have been omitted).



After one layer of the message passing model, we get the following model $\mathcal{I}[\mathcal{MP}_1]$:



After a second layer, we get $\mathcal{I}[\mathcal{MP}_2]$:



The remarkable property of this action model is that it preserves the topology of the input model. This is a well-known fact in distributed computing [64], reformulated here in terms of DEL.

Proposition 3.40. *Let $\mathcal{I} = \langle V, S, \chi, \ell \rangle$ be an input model, and $\mathcal{MP}_N = \langle T, \sim, \mathsf{pre} \rangle$ be the N-layer action model. Then, the product update simplicial model $\mathcal{I}[\mathcal{MP}_N]$ is a subdivision of $\mathcal{I}$, where each edge is subdivided into 3^N edges.*

Usual presentation in terms of local views. This presentation of the layered message-passing model is very unusual from a distributed computing perspective. It does not refer to the very central notion of *view* (or local states); instead, it axiomatizes what it means for two executions to be indistinguishable for one process. To convince ourselves that this axiomatization is correct, let us define the usual protocol complex using local views, and prove that we do get the same result as $\mathcal{I}[\mathcal{MP}_N]$.

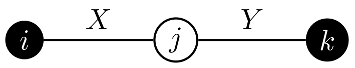
First, let us define by induction on N the *view* of a process in an N-layer execution, starting from the input configuration $X \in \mathcal{F}(\mathcal{I})$ where b has value i and w has value j. For $N = 0$, we define $\mathsf{view}_b(\varnothing, X) = i$ and $\mathsf{view}_w(\varnothing, X) = j$. Let αx be an N-layer execution, where α is an $(N-1)$-layer execution and $x \in \{b, w, \perp\}$. Assume that, after the execution α has occurred, the view of b is $V_b := \mathsf{view}_b(\alpha, X)$, and the view of w is $V_w := \mathsf{view}_w(\alpha, X)$. Then:

– if $x = b$, then $\mathsf{view}_b(\alpha x, X) = (V_b, V_w)$ and $\mathsf{view}_w(\alpha x, X) = (\square, V_w)$.
– if $x = w$, then $\mathsf{view}_b(\alpha x, X) = (V_b, \square)$ and $\mathsf{view}_w(\alpha x, X) = (V_b, V_w)$.
– if $x = \perp$, then $\mathsf{view}_b(\alpha x, X) = (V_b, V_w)$ and $\mathsf{view}_w(\alpha x, X) = (V_b, V_w)$.

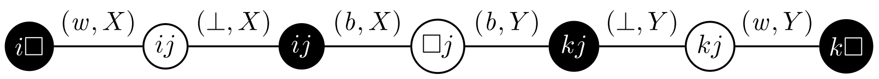
In a pair such as (V_b, V_w), the first component is the message received from process b, and the second component is the message received from process w. The symbol $\square$ indicates that no message was received. A process always “receives” its own message, i.e., it remembers its previous view).

Finally, the protocol complex $\mathcal{P}_N$ of the N-layer message-passing protocol with input complex $\mathcal{I}$ has vertices of the form $(a, \mathsf{view}_a(\alpha, X))$ with $a \in \{b, w\}$, $\alpha \in \{b, w, \perp\}^N$ and $X \in \mathcal{F}(\mathcal{I})$. Its facets are of the form $\{(b, \mathsf{view}_b(\alpha, X)), (w, \mathsf{view}_w(\alpha, X))\}$, with $\alpha \in \{b, w, \perp\}^N$ and $X \in \mathcal{F}(\mathcal{I})$. Thus, a vertex of color a belongs to two such facets whenever $\mathsf{view}_a(\alpha, X) = \mathsf{view}_a(\beta, Y)$, for some α, β, X, Y.

Example 3.41. Suppose that the input complex $\mathcal{I}$ has two facets X and Y, as represented below,



then the protocol complex $\mathcal{P}_1$ for the single-layer protocol is the following one:



The following lemma states that two different choices for the pair (α, X) will necessarily give rise to two distinct facets in $\mathcal{P}$.

Lemma 3.42. *For all $N \in \mathbb{N}$, $\alpha, \beta \in \{\bot, b, w\}^N$, and $X, Y \in \mathcal{F}(\mathcal{I})$, if $\mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y)$ and $\mathsf{view}_w(\alpha, X) = \mathsf{view}_w(\beta, Y)$, then $\alpha = \beta$ and $X = Y$.*

Proof. By induction on N. $\square$ $\square$

Proposition 3.43. *The protocol graph $\mathcal{P}_N$ for the N-layer message-passing protocol is isomorphic to the product update model $\mathcal{I}[\mathcal{MP}_N]$ of Definition 3.39.*

Proof. The main property that we need to prove is the following claim, for any number of layers N, for all $\alpha, \beta \in L_N$ and $X, Y \in \mathcal{F}(\mathcal{I})$:

$$\mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y) \iff (\alpha, X) \sim_b (\beta, Y) \tag{3.1}$$

and similarly for view_w and $\sim_w$. We prove it by induction on N.

– For $N = 0$, this holds from the definitions.
– Let αx and βy be executions of length $N + 1$, with $x, y \in \{b, w, \bot\}$.

(⇐): Assume $(\alpha x, X) \sim_b (\beta y, Y)$. By definition, there are two possible cases.

- $x = y = w$ and $(\alpha, X) \sim_b (\beta, Y)$. By induction hypothesis, we get $\mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y)$. Let us write V_b for this view. Then, by definition, we get $\mathsf{view}_b(\alpha x, X) = (V_b, \square)$ and $\mathsf{view}_b(\beta y, Y) = (V_b, \square)$, which are equal.
- $x, y \in \{\bot, b\}$ and $X = Y$ and $\alpha = \beta$. We write $V_b = \mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y)$, and $V_w = \mathsf{view}_w(\alpha, X) = \mathsf{view}_w(\beta, Y)$. Then by definition $\mathsf{view}_b(\alpha x, X) = (V_b, V_w) = \mathsf{view}_b(\beta y, Y)$.

(⇒): Assume $\mathsf{view}_b(\alpha x, X) = \mathsf{view}_b(\beta y, Y)$. Since a view can never be just '$\square$' (it is either an input value, or a pair of values), we can never have $V_w = \square$. Thus the equality can be true only in one of the two following cases.

- Either $x = y = w$. In which case, we get $\mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y)$ by identifying the first components of the views, and by induction hypothesis, $(\alpha, X) \sim_b (\beta, Y)$. Therefore, $(\alpha x, X) \sim_b (\beta y, Y)$.
- Or $x, y \in \{\bot, b\}$. By identifying the components of the pair, we get $\mathsf{view}_b(\alpha, X) = \mathsf{view}_b(\beta, Y)$ and $\mathsf{view}_w(\alpha, X) = \mathsf{view}_w(\beta, Y)$. By Lemma 3.42 we deduce $\alpha = \beta$ and $X = Y$, and thus we obtain $(\alpha x, X) \sim_b (\beta y, Y)$.

The proof for the correspondence between view_w and $\sim_w$ is similar, with the role of b and w reversed.

Once we have shown (3.1), proving the proposition is just a matter of unfolding the definitions of the product update model. A vertex of $\mathcal{I}[\mathcal{MP}_N]$ is formally given by a pair (v, E), where v is a vertex of $\mathcal{I}$ (say, of color b) and E is an equivalence class of $\sim_b$. Let $(\alpha, X) \in E$ be an action in E. Then, to the vertex (v, E) of $\mathcal{I}[\mathcal{MP}_N]$ we associate the vertex $(b, \mathsf{view}_b(\alpha, X))$ of $\mathcal{P}_N$; this is well-defined thanks to (3.1). Surjectivity is obvious. To show injectivity, assume that (v, E) and (v', E') give the same view. By (3.1), we get $E = E'$; and $v = v'$ according to the precondition pre of the action model. Finally, in $\mathcal{I}[\mathcal{MP}_N]$, we get an edge between (v_b, E_b) and (v_w, E_w) whenever there is an action $t = (\alpha, X)$ with $X := \{v_b, v_w\}$ and $t \in E_b$ and $t \in E_w$. Then by definition of our bijection between the vertices, this edge is sent to the pair $\{(b, \mathsf{view}_b(\alpha, X)), (w, \mathsf{view}_w(\alpha, X))\}$, which is also an edge. $\square$ $\square$

The immediate-snapshot action model

We describe here the *immediate snapshot* action model $\mathcal{IS}$ for one communication exchange among $n+1$ asynchronous agents. As an action model, it is new and to the best of our knowledge it has not been studied from the DEL perspective. Immediate snapshot operations are important in distributed computing, and many variants of computational models based on them have been considered, including multi-round communication exchanges, see e.g. [64, 4]. For the point we want to make about using DEL, the main issues can be studied with this action model, even in the one communication exchange case. In the case of two processes, the immediate-snapshot action model coincides with the layered message-passing model that we presented previously.

The situation we want to model is the following. The $n+1$ agents correspond to $n+1$ concurrent processes. Initially, each process has some input value, and they communicate (only once) through a shared memory array in order to try to learn each other's input values. They use the following protocol: each process has a dedicated memory cell in the array, to which it writes its input value. Then, it takes a snapshot, that is, it reads the whole shared array atomically, in order to see which other input values have been written. Finally, we only take into account the executions of this protocol which satisfy some "immediacy" condition.

In Chapter 1, this protocol was formally defined using carrier maps in Example 1.26. Here, we define it using an action model. We will not try to axiomatize abstractly the indistinguishability relation between immediate-snapshot executions, as we did in Definition 3.39 for the layered message-passing action model. Instead, we directly define it in the usual "distributed computing" way, using the notion of local view. By definition, two immediate-snapshot executions are indistinguishable by some agent whenever they give rise to the same local view for this agent. So, the property (3.1) that appeared in the proof of Proposition 3.43 will be our definition of the indistinguishability relation $\sim$.

An *immediate-snapshot execution* for the set of agents Ag is given by a sequence $c_1, c_2, \ldots, c_m$ of non-empty, disjoint subsets of Ag, whose union is equal to Ag. Such a sequence is called a *sequential partition*. Each c_i is called a *concurrency class*. Notice that $1 \leq m \leq |\mathrm{Ag}|$, and when $m = 1$ all agents take an immediate snapshot concurrently, while if $m = |\mathrm{Ag}|$, all agents take immediate snapshots sequentially. The agents in a concurrency class c_j learn the input values of all the agents in earlier concurrency classes c_i for $i \leq j$, and they also learn which agent wrote which value. In particular, agents in c_m learn the inputs of all the other agents (and there is always at least one such agent). If $m = 1$, then all agents learn all the values.

Example 3.44. As a concrete example, suppose we have four agents $\mathrm{Ag} = \{a, b, c, d\}$, and suppose that their input values are, respectively, 1, 2, 3 and 4. We consider the execution $x = \{b\}, \{a, d\}, \{c\}$ with three concurrency classes. Initially, the shared array is empty, which we represent as $\begin{array}{|c|c|c|c|}\hline \bot & \bot & \bot & \bot \\ \hline\end{array}$, where the four cells correspond to a, b, c, d, in that order. First, process b, which is alone in the first concurrency class, writes its value and immediately reads. Thus, b sees only its own value, and its view is $\begin{array}{|c|c|c|c|}\hline \bot & 2 & \bot & \bot \\ \hline\end{array}$. Then, in the second concurrency class, both a and d write their values simultaneously, and after that, they read simultaneously. So, both a and d have the same view, which is the following array: $\begin{array}{|c|c|c|c|}\hline 1 & 2 & \bot & 4 \\ \hline\end{array}$. Finally, process c goes last in the third concurrency class, and it sees all the values of all the other processes: $\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline\end{array}$.

Notice that, to define the view, we need to know not only the execution $c_1, \ldots, c_m$, but also the input values of the agents. Thus, the action model $\mathcal{IS}$ that we are defining will actually be parameterized by an *input model*, which represents all the possible input values that we want to take into consideration. An *action* of $\mathcal{IS}$ will consist of an execution $c_1, \dots, c_m$, along with an assignment of an input value to each agent. This is in accordance with what is usually done in DEL: for instance, in Example 3.8, to model a situation where an agent reveals its card, we did not have only one action "reveal card", but many actions "reveal that the card is x", for each possible value of x. Then the preconditions make sure that such an action happens in a world where the value of the card is actually x.

To make things simpler, let us fix one particular input model: the simplicial model of Example 3.21 where three agents $\mathrm{Ag} = \{b, g, w\}$ each have a binary input value 0 or 1. Let $\mathcal{I} = \langle V_{\mathcal{I}}, S_{\mathcal{I}}, \chi, \ell\rangle$ be the corresponding simplicial model, and denote a facet $X \in \mathcal{F}(\mathcal{I})$ by a binary sequence $b_0b_1b_2$, corresponding to the three values of b, g, w, in that order (alphabetical). We take the set of atomic propositions $\mathrm{At} = \{\mathsf{input}_a^x \mid a \in \mathrm{Ag}, x \in \{0,1\}\}$, where input_a^x is the atomic proposition expressing that agent a has input value x.

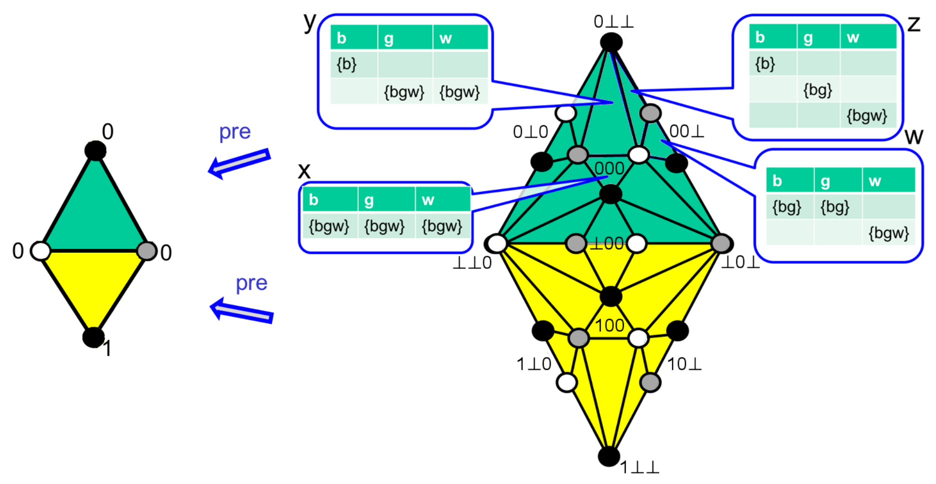
Definition 3.45. We define the *immediate snapshot* action model $\mathcal{IS} = \langle T, \sim, \mathsf{pre}\rangle$ as follows. An action $t \in T$ is given by the data c, b_0, b_1, b_2, where c is a sequential partition of $\mathrm{Ag} = \{b, g, w\}$, and b_0, b_1, b_2 are binary values 0 or 1. Such an action will be written $c^{b_0b_1b_2}$. The precondition $\mathsf{pre}(c^{b_0b_1b_2})$ of this action is a formula expressing the fact that the inputs of the agents b, g, w are respectively b_0, b_1, b_2. Therefore, $\mathsf{pre}(c^{b_0b_1b_2})$ is true precisely in the facet $b_0b_1b_2$ of $\mathcal{I}$. Formally, we can take the formula $\mathsf{pre}(c^{b_0b_1b_2}) = \mathsf{input}_b^{b_0} \wedge \mathsf{input}_g^{b_1} \wedge \mathsf{input}_w^{b_2}$. The indistinguishability relation is defined as $t \sim_a t'$ iff $\mathsf{view}_a(t) = \mathsf{view}_a(t')$, where $\mathsf{view}_a(t)$ is defined as expected: if $c = c_1, \dots, c_m$ is a sequential partition of Ag, and the agent a is in c_j, then $\mathsf{view}_a(c^{b_0b_1b_2})$ is the vector obtained from $b_0b_1b_2$ by replacing the value b_i by $\bot$ whenever the corresponding agent is not in $\bigcup_{i\leq j} c_i$.

We can also describe the *simplicial action model* $G(\mathcal{IS})$, which is the simplicial counterpart of $\mathcal{IS}$ given by Theorem 3.16. It contains exactly the same data as $\mathcal{IS}$, but it is translated into the language of chromatic simplicial complexes, which allows us to visualise it better. Formally, we have $G(\mathcal{IS}) = \langle V_T, S_T, \chi_T, \mathsf{pre}\rangle$ where $\langle V_T, S_T, \chi_T\rangle$ is a chromatic simplicial complex whose vertices are $V_T = \{(a, \mathsf{view}_a(c^{b_0b_1b_2})) \mid a \in \mathrm{Ag}, c \text{ is an execution, and } b_0, b_1, b_2 \in \{0,1\}\}$; and whose facets are of the form:

$$X = \{(b, \mathsf{view}_b(c^{b_0b_1b_2})), (g, \mathsf{view}_g(c^{b_0b_1b_2})), (w, \mathsf{view}_w(c^{b_0b_1b_2}))\}$$

The precondition of such a facet is $\mathsf{pre}(X) = \mathsf{input}_b^{b_0} \wedge \mathsf{input}_g^{b_1} \wedge \mathsf{input}_w^{b_2}$.

The picture below illustrates (part of) the simplicial action model $G(\mathcal{IS})$. The two triangles on the left represent two facets of the input model $\mathcal{I}$, with input values 000 (green) and 100 (yellow). On the right are the corresponding facets of $G(\mathcal{IS})$. We recognize the standard chromatic division of the input complex, where each of the two input triangles have been subdivided into 13 smaller triangles: one for each possible sequential partition of $A = \{b, g, w\}$. Four of these sequential partitions are depicted in the bubbles X, Y, Z, W. The tables in the bubbles show the scheduling of the execution from top to bottom: for example, in execution Z, process b goes first and sees only itself; then process g goes second and sees both b and g; then process w goes last and sees everyone. The colors black, grey, white of the vertices correspond respectively to agents b, g, w. The view of each vertex is written next to it; when two (or three) neighboring vertices have the same view, it is written only once, on the edge (or triangle) between the two (or three) vertices. The precondition of all the green facets on the right is true exactly in the green facet of the input model $\mathcal{I}$ on the left, and similarly for the yellow facets.



When we compute the product update model $\mathcal{I}[\mathcal{IS}]$, we obtain a simplicial model whose underlying simplicial complex is the same as the one of $\mathcal{IS}$, depicted on the right. So, starting from the input model $\mathcal{I}$, the effect of applying $\mathcal{IS}$ is to subdivide each facet of the input. The same thing happens for any input model $\mathcal{I}$. So, the simplicial model $\mathcal{I}[\mathcal{IS}]$ is isomorphic to the usual immediate-snapshot protocol complex (Example 1.26), as expected. Remarkably, the topology of the input simplicial complex is preserved: if $\mathcal{I}$ is a sphere as in Example 3.21, then $\mathcal{I}[\mathcal{IS}]$ is still a sphere.

In the rest of the chapter, since by Proposition 3.36, $\mathcal{I}[G(\mathcal{IS})]$ and $\mathcal{I}[\mathcal{IS}]$ are isomorphic, we drop the distinction between regular and simplicial action models, and just write $\mathcal{IS}$ for both of them.

Multi-round communication In the $\mathcal{IS}$ model, each agent executes a single immediate snapshot. Iterating this model gives rise to the *iterated immediate snapshot model* $\mathcal{IS}^r$ [64, 107], where each agent executes r consecutive immediate snapshots, on r consecutive shared memory arrays. Starting from an input model $\mathcal{I}$, the effect of applying the iterated immediate snapshot protocol is to subdivide each facet of the input complex r times. So, once again, the topology of $\mathcal{I}$ is preserved in $\mathcal{I}[\mathcal{IS}^r]$. In the non-iterated version, the r immediate snapshots are executed on the same memory. A subdivision is still obtained, but it is more complex [4]. If all schedules are considered, not only immediate snapshot schedules, then the topology is still preserved, even-though the resulting complex is no longer a subdivision, see e.g. [9] for more precise meaning about these claims and further discussion.

3.3.2 Tasks as action models

We now show how a concurrent *task* can be specified using an action model. As before, we suppose fixed an input simplicial model $\mathcal{I}$, describing all the possible initial configurations, where one input value is given to each agent. The agents communicate with each other according to some protocol action model, such as $\mathcal{MP}_N$ or $\mathcal{IS}$ of Section 3.3.1. Then, based on the information that it acquired after communication, the agent produces an output value. A task specifies the output values that the agents may decide, when starting in a given input configuration. Thus, in this section, we use DEL in a novel way. Indeed, we do not use it to model an exchange of information between agents: we use DEL as a way to provide a specification of the problem that the agents should solve.

Consider a simplicial model $\mathcal{I} = \langle V, S, \chi, \ell \rangle$ called the *input simplicial model*. Each facet of $\mathcal{I}$, with its labeling ℓ, represents a possible initial configuration.

Definition 3.46. A *task* for $\mathcal{I}$ is an action model $\mathcal{T} = \langle T, \sim, \mathsf{pre} \rangle$ for agents Ag, where each action $t \in T$ consists of a function $t : \mathrm{Ag} \to V^{out}$, where V^{out} is an arbitrary domain of *output values*. Such an action is interpreted as a possible assignment of an output value for each agent. Each such t has a precondition that is true in one or more facets of $\mathcal{I}$, interpreted as "if the input configuration is a facet in which $\mathsf{pre}(t)$ holds, and every agent $a \in \mathrm{Ag}$ decides the value $t(a)$, then this is a valid execution". The indistinguishability relation is defined as $t \sim_a t'$ when $t(a) = t'(a)$.

Remark 3.47. This way of specifying a task is weaker than the usual distributed computing definition based on carrier maps (Definition 1.19). Indeed, it only allows us to specify the input/output relation on $|\mathrm{Ag}|$-tuples. Hence, we cannot express what the task specification should be for sub-executions where some processes do not participate. That is, we can only specify tasks *without crashes*. See also Section 1.1.3 for a discussion of this subtle difference between the two ways of specifying a task.

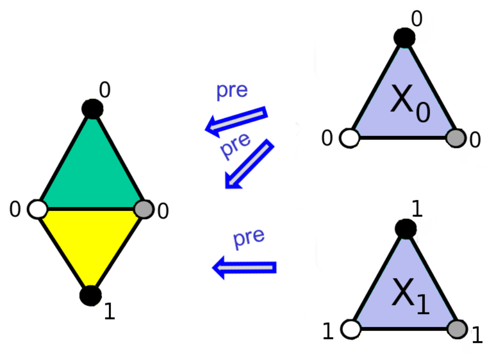
Example 3.48. The most important task in distributed computing is *binary consensus*. Assume we have three agents $\mathrm{Ag} = \{b, g, w\}$, and the input model $\mathcal{I}$ is the binary input model from Example 3.21. This means that every combination of 0's and 1's is a possible initial configuration. At the end of the computation, each agent must decide an output value, which can be either 0 or 1. We write d_b, d_g, d_w for the decision value of agent b, g, w respectively. The goal of the task is to achieve the following properties:

– *Agreement:* the agents must decide the same value, i.e. $d_b = d_g = d_w$.

– *Validity:* the agreed value must be one of the three inputs.

Say, for example, that the three inputs are $(0, 1, 0)$, then the three outputs can be either $(1, 1, 1)$ or $(0, 0, 0)$. On the other hand, if the three inputs are $(1, 1, 1)$, then the only possible output is $(1, 1, 1)$, because agreeing on value 0 would break the validity condition.

Formally, this task is described by an action model $\mathcal{T} = \langle T, \sim, \mathsf{pre} \rangle$. There are two possible combinations of outputs: t_0 where all the decisions are 0, and t_1 where all the decisions are 1. There is no indistinguishability relation between these two actions. The precondition $\mathsf{pre}(t_0)$ must be true in all the facets of $\mathcal{I}$ where the output $(0, 0, 0)$ is valid, i.e., whenever at least one of the agents has input value 0. If input_a^x denotes the atomic formula saying that agent a has input value x, we take $\mathsf{pre}(t_0) = \mathsf{input}_b^0 \vee \mathsf{input}_g^0 \vee \mathsf{input}_w^0$. Similarly, $\mathsf{pre}(t_1)$ is true whenever at least one agent has input 1, i.e., we take $\mathsf{pre}(t_1) = \mathsf{input}_b^1 \vee \mathsf{input}_g^1 \vee \mathsf{input}_w^1$. The picture below represents the associated simplicial actions model $G(\mathcal{T})$ (in blue). It has two disjoint facets, X_0 where all decisions are 0, and X_1 where all decisions are 1. The two simplices on the left are a fragment of the input model $\mathcal{I}$.



Notice that the simplicial action model $G(\mathcal{T})$ is isomorphic to the output complex of the consensus task described in Example 1.20. However, when we compute the simplicial model $\mathcal{I}[\mathcal{T}]$, we get a new simplicial complex which is usually not considered in distributed computing. Recall that the binary input model $\mathcal{I}$ is a triangulated sphere. Then, $\mathcal{I}[\mathcal{T}]$ consists of two disjoint copies of $\mathcal{I}$, obtained via the cartesian products $\mathcal{I} \times X_0$ and $\mathcal{I} \times X_1$, except that one simplex is missing from each sphere. In the sphere $\mathcal{I} \times X_0$, the simplex corresponding to the $(1, 1, 1)$ is removed, because it does not satisfy the precondition of X_0. In the sphere $\mathcal{I} \times X_1$, the simplex $(0, 0, 0)$ is missing. Intuitively, the simplicial model $\mathcal{I}[\mathcal{T}]$ represents the minimal amount of knowledge that the processes should acquire in order to solve the task $\mathcal{T}$.

3.3.3 DEL definition of task solvability

Suppose fixed an input simplicial model $\mathcal{I}$, and a protocol action model $\mathcal{A}$ (such as $\mathcal{IS}$ or $\mathcal{MP}_N$). We get the *protocol simplicial model* $\mathcal{I}[\mathcal{A}]$, which represents the knowledge gained by the agents after executing $\mathcal{A}$. To solve a task $\mathcal{T}$, each agent, based on its own knowledge, should produce an output value, such that the vector of output values respects the specification of the task.

The following gives a formal epistemic logic semantics to task solvability. Recall that the product update model $\mathcal{I}[\mathcal{A}]$ is a sub-complex of the cartesian product $\mathcal{I} \times \mathcal{A}$, whose vertices are of the form (i, t) with i a vertex of $\mathcal{I}$ and t a vertex of $\mathcal{A}$. We write $\pi_{\mathcal{I}}$ for the first projection on $\mathcal{I}$, which is a morphism of simplicial models.

Definition 3.49. A task $\mathcal{T}$ is *solvable* by the action model $\mathcal{A}$ if there exists a morphism $\delta : \mathcal{I}[\mathcal{A}] \to \mathcal{I}[\mathcal{T}]$ such that $\pi_I \circ \delta = \pi_I$, i.e., the diagram of simplicial complexes below commutes.

$$\begin{tikzcd}
 & \mathcal{I}[\mathcal{A}] \arrow[dl, "\pi_{\mathcal{I}}"'] \arrow[d, dashed, "\delta"] \\
\mathcal{I} & \mathcal{I}[\mathcal{T}] \arrow[l, "\pi_{\mathcal{I}}"]
\end{tikzcd}$$

The justification for this definition is the following. A facet X in $\mathcal{I}[\mathcal{A}]$ corresponds to a pair (I, t), where $I \in \mathcal{F}(\mathcal{I})$ represents input value assignments to all agents, and $t \in \mathcal{A}$ represents an action, codifying the communication exchanges that took place. The morphism δ takes X to a facet $\delta(X) = (I, dec)$ of $\mathcal{I}[\mathcal{T}]$, where $dec \in \mathcal{T}$ is the assignment of decision values that the agents will choose in the situation X. Moreover, $\mathsf{pre}(dec)$ holds in I, meaning that dec corresponds to valid decision values for inputs I. The commutativity of the diagram expresses the fact that both X and $\delta(X)$ correspond to the same input assignment I. Now consider a single vertex $v \in X$ with $\chi(v) = a \in \mathrm{Ag}$. Then, agent a decides its value solely according to its knowledge in $\mathcal{I}[\mathcal{A}]$: if another facet X' contains v, then $\delta(v) \in \delta(X) \cap \delta(X')$, meaning that a has to decide the same value in both situations.

The diagram above has two illuminating interpretations. First, by Lemma 3.15, we know that the knowledge about the world of each agent can only decrease (or stay constant) along the δ arrow. So agents should improve knowledge through communication, by going from $\mathcal{I}$ to $\mathcal{I}[\mathcal{A}]$. The task is solvable if and only if there is enough knowledge in $\mathcal{I}[\mathcal{A}]$ to match the knowledge required by $\mathcal{I}[\mathcal{T}]$. Secondly, the possibility of solving a task depends on the existence of a certain simplicial map from the complex of $\mathcal{I}[\mathcal{A}]$ to the complex of $\mathcal{I}[\mathcal{T}]$. Recall that a simplicial map is the discrete equivalent of a continuous map, and hence task solvability is of a topological nature.

Outline of impossibility proofs. To prove that a task $\mathcal{T}$ is not solvable in $\mathcal{A}$, our usual proof method goes like this. Assume $\delta : \mathcal{I}[\mathcal{A}] \to \mathcal{I}[\mathcal{T}]$ exists, then:

1. Pick a well-chosen positive epistemic logic formula φ,

2. Show that φ is true in every world of $\mathcal{I}[\mathcal{T}]$,
3. Show that there exists a world X of $\mathcal{I}[\mathcal{A}]$ where φ is false,
4. By Lemma 3.15, since φ is true in $\delta(X)$ then it must also be true in X, which is a contradiction with the previous point.

This kind of proof is interesting because it explains the reason why the task is not solvable. The formula φ represents some amount of knowledge which the processes must acquire in order to solve the task. If φ is given, the difficult part of the proof is usually the third point: finding a world X in the protocol complex where the processes did not manage to obtain the required amount of knowledge. The existence of this world can be proved using theorems of combinatorial topology, such as Sperner's lemma or the Index lemma. Thus, the formula φ describes the epistemic content of the abstract topological arguments for unsolvability. For example, when the usual topological proof would claim that consensus is not solvable because of the connectedness of the protocol complex, we will see in the next section that another reason for impossibility is that the processes did not reach common knowledge of the set of input values.

3.4 Examples

In this section, we use our DEL setting to analyze the solvability in the immediate-snapshot model of three well-studied distributed computing tasks: consensus, approximate agreement, and set agreement. Their solvability is already well-understood; in particular, Theorems 3.50, 3.54 and 3.55 are already known. The novelty here is that we give new proofs based on logical arguments.

3.4.1 Consensus

Let $\mathcal{I} = \langle V, S, \chi, \ell \rangle$ be the initial simplicial model for binary input values (see Example 3.21), and $\mathcal{T} = \langle V_T, S_T, \chi_T, \mathsf{pre} \rangle$ be the simplicial action model for binary consensus (see Example 3.48). Thus, $\mathcal{T}$ has only two facets, X_0 where all decisions are 0 and X_1, where all decisions are 1. The underlying complex of $\mathcal{I}[\mathcal{T}]$ consists of two disjoint simplicial complexes: $I_0 \times X_0$ and $I_1 \times X_1$, where I_0 consists of all input facets with at least one 0, and I_1 consists of all input facets with at least one 1. Notice that, in fact, each of the two complexes $I_i \times X_i$, for $i \in \{0, 1\}$, is isomorphic to I_i, since X_i consists of just one facet. The labeling $\ell[\mathcal{T}]$ of the vertices of $\mathcal{I}[\mathcal{T}]$ is copied from the first component I_i.

To show that binary consensus cannot be solved by the immediate snapshot protocol, we must prove that the map $\delta : \mathcal{I}[\mathcal{IS}] \to \mathcal{I}[\mathcal{T}]$ of Definition 3.49 does not exist. The usual proof of impossibility uses a topological obstruction to the existence of δ. Here, instead, we exhibit a logical obstruction.

Theorem 3.50. *The binary consensus task is not solvable by $\mathcal{IS}$.*

Proof. Following the outline for impossibility proofs of Section 3.3.3, the first step is to find a formula φ expressing the knowledge which is required to solve the task. Let $\varphi_i := \bigvee_{a \in \mathrm{Ag}} \mathsf{input}_a^i$ be a formula denoting that at least one agent has input i. We pick the formula $\varphi := C_{\mathrm{Ag}}\, \varphi_0 \vee C_{\mathrm{Ag}}\, \varphi_1$.

For $i \in \{0, 1\}$, it is straightforward to check that at any facet Y of $I_i \times X_i$, there is common knowledge that at least one input is i, that is, $\mathcal{I}[\mathcal{T}], Y \models C_{\mathrm{Ag}} \varphi_i$. So, the formula φ is true in every world of $\mathcal{I}[\mathcal{T}]$. Now, consider the simplicial model $\mathcal{I}[\mathcal{IS}]$, for the immediate snapshot action model. Since $\mathcal{I}[\mathcal{IS}]$ is a subdivision of $\mathcal{I}$, it is connected, and therefore from any facet X of $\mathcal{I}[\mathcal{IS}]$, there is a path to the facet where all inputs are 0, and to the facet where all inputs are 1. Hence, we gave $\mathcal{I}[\mathcal{IS}], X \not\models C_A\varphi_i$, for both $i = 0$ and $i = 1$. That is, $\mathcal{I}[\mathcal{IS}], X \not\models \varphi$.

Finally, we know that morphisms of simplicial models cannot "gain knowledge about the world" from Lemma 3.15, and so, there cannot be a morphism δ from $\mathcal{I}[\mathcal{IS}]$ to $\mathcal{I}[\mathcal{T}]$, by the two previous claims. □

Remark 3.51. There are two things to observe about this proof. First, notice that the argument holds for any other model, instead of $\mathcal{IS}$, which is connected. For instance, this is the case for any number of immediate-snapshot communication rounds by wait-free asynchronous agents [62]. Secondly, the usual topological argument for impossibility is the following: because simplicial maps preserve connectedness, δ cannot send a connected simplicial complex into a disconnected simplicial complex. Notice how in both the logical and the topological proofs, the main ingredient is a connectedness argument.

3.4.2 Approximate agreement

We now discuss a weaker version of the consensus task, where agents are required to decide values which are close to each other, not necessarily equal. It turns out, that no matter how close to each other one requires the agents to decide, this task is solvable in the immediate snapshot (multi-round) model. Many versions of this task have been considered. We present here a simple one, for two agents, b and w.

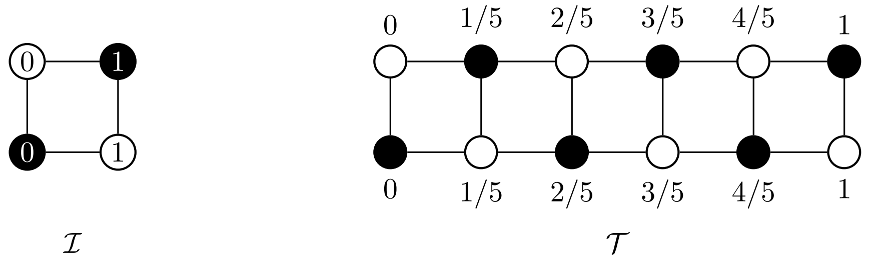
The input complex is the binary input complex for two agents, depicted on the left of Example 3.21: so, every possible combination of 0 and 1 can be assigned to the two agents. Their goal will be to output real values in the interval $[0, 1]$, such that:

– if both inputs are the same (i.e., 00 or 11), they should both choose their input value as output.

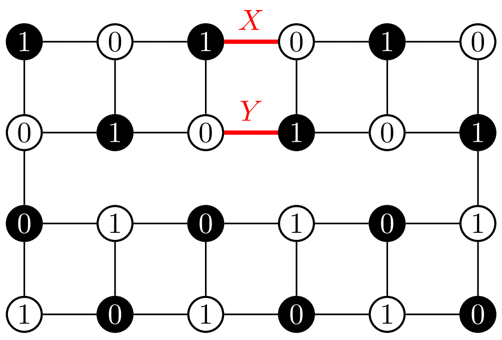
– if the inputs are different (i.e., 01 or 10), they should decide values d_b and d_w such that $|d_b - d_w| \leq \varepsilon$.

In order to be able to work with finite models, we define a discrete version of this task, *M-approximate agreement*. The output values are only allowed to be of the form k/M for $0 \leq k \leq M$. The two decision values should be within $1/M$ of each other: $|d_b - d_w| \leq 1/M$.

Let $\mathcal{I} = \langle V, S, \chi, \ell\rangle$ the binary input model for two agents, and $\mathcal{T} = \langle V_\mathcal{T}, S_\mathcal{T}, \chi_\mathcal{T}, \mathsf{pre}\rangle$ the following simplicial action model. The set of vertices of $\mathcal{T}$ is $V_\mathcal{T} = \{(a, k/M) \mid a \in \mathrm{Ag} \text{ and } 0 \leq k \leq M\}$. The facets of $\mathcal{T}$ are edges $X_{k,k'} = \{(b, k/M), (w, k'/M)\}$ with $|k - k'| \leq 1$. The color of a vertex is $\chi(a, k/M) = a$. The precondition $\mathsf{pre}(X_{0,0})$ is true in the worlds 00, 01 and 10 of $\mathcal{I}$; the precondition $\mathsf{pre}(X_{M,M})$ is true in the worlds 11, 01 and 10; and all the other preconditions $\mathsf{pre}(X_{k,k'})$ are true in the worlds 01 and 10. In the figure below are depicted the input model $\mathcal{I}$ (left) and the simplicial action model $\mathcal{T}$ (right), for $M = 5$.



The product update $\mathcal{I}[\mathcal{T}]$ is the simplicial model depicted in the next figure. The numbers depicted in the nodes are the atomic propositions describing the input values from $\mathcal{I}$. The decisions values (of the form $k/5$) are implicit, the first column of nodes corresponds to the decision value 0, the second column is decision value $1/5$, and so on. For example, the world marked X on the figure corresponds to the situation where b started with value 1 and decided value $2/5$, while w started with value 0 and decided value $3/5$. This is a correct execution of the 5-approximate agreement task.



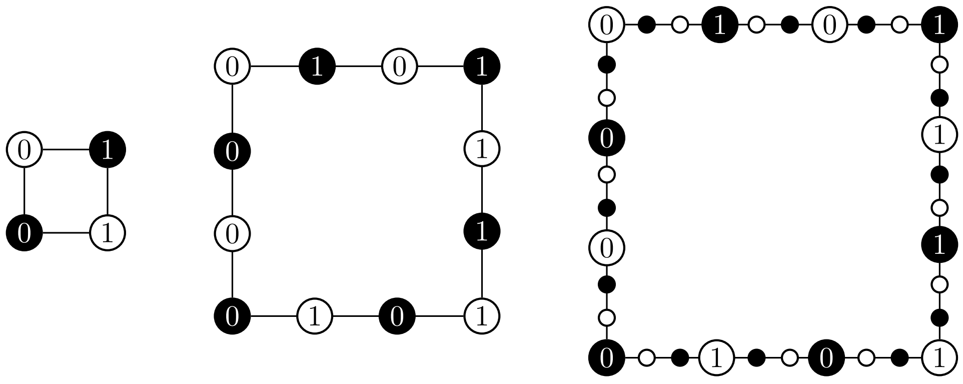
$\mathcal{I}[\mathcal{T}]$

Notice that the world X on the figure is the one with the most knowledge in the following sense. We write φ_{01} the formula expressing that the two inputs are different, and $E\varphi = K_b\varphi \wedge K_w\varphi$ for the group knowledge of φ among the agents $\{b, w\}$. Then, we have $\mathcal{I}[\mathcal{T}], X \models E^3\varphi_{01}$, where E^3 denotes three nested E operators. The world Y is one step closer from the 00 or 11 edges, so $\mathcal{I}[\mathcal{T}], Y \not\models E^3\varphi_{01}$, but we have the weaker knowledge $\mathcal{I}[\mathcal{T}], Y \models E^2\varphi_{01}$.

Lemma 3.52. *In the simplicial model $\mathcal{I}[\mathcal{T}]$ for the M-approximate agreement task, there is a world X such that $\mathcal{I}[\mathcal{T}], X \models E^k\varphi_{01}$, for $k = \lceil M/2 \rceil$.*

Proof. We choose for X one of the "middle" worlds, where the two agents have input 0 and 1, and they decide the values k/M and $(k-1)/M$, for $k = \lceil M/2 \rceil$. ☐

We now study the solvability of this task in the N-layer message-passing model $\mathcal{MP}_N$. Each consecutive layer subdivides each edge into three parts. The picture below shows the input model $\mathcal{I}$, the model $\mathcal{I}[\mathcal{MP}_1]$ after one layer, and the model $\mathcal{I}[\mathcal{MP}_2]$ after two layers.



$\mathcal{I}$ $\mathcal{I}[\mathcal{MP}_1]$ $\mathcal{I}[\mathcal{MP}_2]$

Lemma 3.53. *In the N-layer message-passing model $\mathcal{I}[\mathcal{MP}_N]$, there is no world X where the formula $E^k\varphi_{01}$ is true for $k = \lceil 3^N/2 \rceil$, i.e., for all X, $\mathcal{I}[\mathcal{MP}_N], X \not\models E^k\varphi_{01}$.*

Proof. After N layers, each of the four edges of the input model $\mathcal{I}$ have been subdivided into 3^N edges. Thus, every world is at a distance at most $k-1$ from the nearest world with inputs 00 or 11. ☐

Putting the two lemmas together, we get the following result:

Theorem 3.54. *The M-approximate agreement task is not solvable in the N-layer message-passing model, when $N < \lceil \log_3(M) \rceil$.*

Proof. Assume by contradiction that the task is solvable. Then, we have a map $\delta : \mathcal{I}[\mathcal{MP}_N] \to \mathcal{I}[\mathcal{T}]$. Our goal is to find a contradiction using Lemma 3.15. To achieve this, we should find a formula φ and a world Z of $\mathcal{I}[\mathcal{MP}_N]$, such that φ is false in Z but true in $\delta(Z)$. We have to be a little bit cautious, because there is no guarantee that the world X exhibited in Lemma 3.52 is in the image of δ.

Since $\mathcal{I}[\mathcal{MP}_N]$ is connected, and simplicial maps preserve connectedness, its image $\delta(\mathcal{I}[\mathcal{MP}_N])$ is connected too. Moreover, the world 00 and the world 11 of $\mathcal{I}[\mathcal{T}]$ must be in the image of δ, because of the commutative diagram of Definition 3.49. By connectedness, one of the two middle worlds X or Y must be in the image of δ. Since Y is one step closer to the edge than X, we "lose" one nested E operator, and we have by Lemma 3.52: $\mathcal{I}[\mathcal{T}], Y \models E^{\lfloor M/2 \rfloor} \varphi_{01}$.

We now show that no world of $\mathcal{I}[\mathcal{MP}_N]$ has this amount of knowledge, which contradicts Lemma 3.15. By Lemma 3.53, the formula $E^{\lceil 3^N/2 \rceil} \varphi_{01}$ is false in every world of $\mathcal{I}[\mathcal{MP}_N]$. Since $N < \lceil \log_3(M) \rceil$ implies $\lceil 3^N/2 \rceil \leq \lfloor M/2 \rfloor$, this concludes the proof. $\square$

Conversely, it is known (and easy to show) in distributed computing that M-approximate agreement is solvable in $\mathcal{MP}_N$ whenever $N \geq \lceil \log_3(M) \rceil$, see [64] for example. The proof of the above theorem sheds light on the required knowledge to solve approximate agreement: while consensus is about reaching common knowledge, approximate agreement is about reaching some finite level of nested knowledge. This is possible to solve as long as we perform a large-enough number of layers.

3.4.3 2-Set agreement

The k-set agreement task (see Example 1.21) is a weaker version of consensus where the agents must agree on at most k different values. From the topological perspective, this class of tasks is interesting because the solvability of this task is related to the $(k-1)$-*connectedness* of the protocol complex [64].

For $k = 1$, the 1-set agreement task is the consensus task. And indeed, as we saw in Section 3.4.1, the unsolvability proof relies on the 0-connectedness (or *path-connectedness*) of the protocol complex. From the epistemic logic perspective, the unsolvability of consensus is related to common knowledge. The link between common knowledge and the path-connectedness of the model is well known to logicians: since it is a 1-dimensional property, it can be expressed using standard Kripke models.

In this section, we propose to study the 2-set agreement task from the point of view of epistemic logic. The reason why 2-set agreement is unsolvable is related to a higher-dimensional topological property: the fact that the protocol complex is *simply-connected*, that is, that any loop on the protocol complex can be continuously contracted to a single point. In other words, this means that there is no "hole" in the protocol complex. This kind of topological property cannot be expressed on a Kripke model; so, to study 2-set agreement from an epistemic logic perspective, the use of simplicial models seems mandatory.

Another important thing to note about this 2-set agreement task is that it is strictly weaker than consensus (since consensus cannot be implemented using set-agreement objects [65]), but strictly stronger than approximate agreement (since set-agreement is not solvable using read/write memory [64]). So, according to what we did in Sections 3.4.1 and 3.4.2, the amount of knowledge required to solve 2-set agreement should be less than common knowledge, but more than any finite number of nested knowledge. As far as we know, no such notion of knowledge has ever been exhibited in the epistemic logic literature.

Let $\mathcal{I} = \langle V, S, \chi, \ell \rangle$ be the input simplicial model for three agents $\mathrm{Ag} = \{b, w, g\}$, and three possible input values, $\{0, 1, 2\}$. This input model was already used in Chapter 1 and is depicted in Figure 1.3. The simplicial action model $\mathcal{T} = \langle V_T, S_T, \chi_T, \mathsf{pre} \rangle$ for 2-set agreement is defined as follows. The vertices of $\mathcal{T}$ are pairs of the form (a, d) with $a \in \mathrm{Ag}$ and $d \in \{0, 1, 2\}$, and the facets of $\mathcal{T}$ are $X_{d_0,d_1,d_2} = \{(b, d_0), (g, d_1), (w, d_2)\}$, for each vector d_0, d_1, d_2, such that $d_i \in \{0, 1, 2\}$ and $|\{d_0, d_1, d_2\}| \leq 2$. The preconditions are $\mathsf{pre}(X_{d_0,d_1,d_2}) = \varphi_{d_0} \wedge \varphi_{d_1} \wedge \varphi_{d_2}$, where the formula φ_i expresses that at least one agent has input i.

The underlying simplicial complex of $\mathcal{T}$ is the "necklace of spheres" on the right of Figure 1.3. But, unlike $\mathcal{I}$, it does not have the six extra simplices depicted on the left. In particular, $\mathcal{T}$ has a 1-dimensional hole in the middle of the "necklace", so it is not simply connected. On the other hand, $\mathcal{I}$ is isomorphic to a wedge of spheres; so, in particular, it is simply connected (see [64] for a formal proof of this fact). Using these topological facts about $\mathcal{I}$ and $\mathcal{T}$, one can prove the following theorem [64, chap. 10].

Theorem 3.55. *The 2-set agreement task is not solvable by $\mathcal{IS}$.*

Our goal here would be to find an epistemic logic formula φ witnessing the unsolvability the task, as explained in Section 3.3.3. Unfortunately, we have not been able to find such a formula. In fact, we conjecture that no suitable formula φ exists for this particular task. Intuitively, the reason why our proof method fails is because the language $\mathcal{L}_{CK}(\mathrm{Ag}, \mathrm{At})$ is too weak to express the amount of knowledge which is required to solve 2-set agreement.

In the next section, we study another task, the *equality negation task* for two processes, which was already mentioned in Section 1.4. For this simpler task, we will actually have a proof that there is no formula φ witnessing the unsolvability. Then, in Section 3.6, we describe a way to strengthen the language of our logic by adding new atomic propositions. This allows us to prove the impossibility of solving both equality negation and 2-set agreement, using a logical formula.

3.5 Limits of the DEL approach

In this section, we exhibit a limit of our DEL approach for finding logical obstructions to the solvability of concurrent tasks, as outlined in Section 3.3.3. Namely, we exhibit a task, which is known to be unsolvable, but which is such that no epistemic logic formula φ can witness the unsolvability of the task. This task will be the *equality negation task*, which was introduced by Lo and Hadzilacos [88]. In Section 1.4, we extended it to n processes and studied its (un)solvability in the immediate-snapshot model. Here, we focus on the original version of the task, for 2-processes. A simple topological proof of impossibility was given in the beginning of Section 1.4. We will try to find an epistemic logic proof of impossibility, in order to understand why it fails.

3.5.1 Bisimulation between simplicial models

First, we define the notion of bisimulation between simplicial models. This is a straightforward translation of the usual notion of bisimulation for Kripke models, using the equivalence of Theorem 3.20.

Definition 3.56 (Bisimulation)**.** Let $\mathcal{M} = \langle V, S, \chi, \ell \rangle$ and $\mathcal{M}' = \langle V', S', \chi', \ell' \rangle$ be two simplicial models. A relation $R \subseteq \mathcal{F}(M) \times \mathcal{F}(M')$ is a *bisimulation* between $\mathcal{M}$ and $\mathcal{M}'$ if the following conditions hold:

(i) If $X \; R \; X'$ then $\ell(X) = \ell'(X')$.

(ii) For all $a \in \mathrm{Ag}$, if $X \; R \; X'$ and $a \in \chi(X \cap Y)$, then there exists $Y' \in \mathcal{F}(M')$ such that $Y \; R \; Y'$ and $a \in \chi'(X' \cap Y')$.

(iii) For all $a \in \mathrm{Ag}$, if $X \; R \; X'$ and $a \in \chi'(X' \cap Y')$, then there exists $Y \in \mathcal{F}(M)$ such that $Y \; R \; Y'$ and $a \in \chi(X \cap Y)$.

When R is a bisimulation and $X \; R \; X'$, we say that X and X' are *bisimilar*.

The next lemma states that two bisimilar worlds satisfy exactly the same formulas. This is a well-known fact in the context of Kripke models. Since we simply translated the usual definition of bisimulation in the language of simplicial models, the same result also holds for simplicial models.

Lemma 3.57. *Let R be a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$. Then for all facets X, X' such that $X \; R \; X'$, and for every epistemic logic formula φ,*

$$\mathcal{M}, X \models \varphi \qquad \textit{iff} \qquad \mathcal{M}', X' \models \varphi$$

Proof. We prove it by induction on φ. Let $X \in \mathcal{F}(M)$ and $X' \in \mathcal{F}(M')$ be facets with $X \; R \; X'$.

– Case $p \in \mathrm{At}$. Since we assumed $X \; R \; X'$, by Definition 3.56(i) this implies that $\ell(X) = \ell'(X')$. Then, $\mathcal{M}, X \models p \iff \mathcal{M}', X' \models p$, because we have $p \in \ell(X) \iff p \in \ell'(X')$.

– Case $\neg\varphi$. By induction hypothesis, $\mathcal{M}, X \models \varphi$ iff $\mathcal{M}', X' \models \varphi$. Then the negations are also equivalent, that is, $\mathcal{M}, X \models \neg\varphi \iff \mathcal{M}', X' \models \neg\varphi$.

– Case $\varphi \wedge \psi$. By induction hypothesis, $\mathcal{M}, X \models \varphi$ iff $\mathcal{M}', X' \models \varphi$ and similarly for ψ. Then we trivially get $\mathcal{M}, X \models \varphi \wedge \psi \iff \mathcal{M}', X' \models \varphi \wedge \psi$.

– Case $K_a \, \varphi$. Assume that $\mathcal{M}, X \models K_a \, \varphi$ for some agent $a \in \mathrm{Ag}$. That is, we have:

$$\forall Y \in \mathcal{F}(M), \quad a \in \chi(X \cap Y) \implies \mathcal{M}, Y \models \varphi$$

We want to show that $\mathcal{M}', X' \models K_a \varphi$. Let Y' be an arbitrary facet of $\mathcal{M}'$ such that $a \in \chi'(X' \cap Y')$. Since $X \; R \; X'$, by Definition 3.56(iii), we know that there exists $Y \in \mathcal{F}(M)$ with $a \in \chi(X \cap Y)$ and $Y \; R \; Y'$. Using the fact that $\mathcal{M}, X \models K_a \, \varphi$ and $a \in \chi(X \cap Y)$, we get $\mathcal{M}, Y \models \psi$. Finally, by induction hypothesis, since $Y \; R \; Y'$, this implies $\mathcal{M}', Y' \models \psi$.

The converse implication is proved similarly, using Definition 3.56(iii) instead.

– Case $C_A \, \varphi$. We assume $\mathcal{M}, X \models C_A \, \varphi$ and show that $\mathcal{M}', X' \models C_A \varphi$. Let $Y' \in \mathcal{F}(\mathcal{M}')$ such that there is a finite path from X' to Y' going through facets which share an A-colored vertex. By iterating the previous reasoning a finite number of times, we will get a facet $Y \in \mathcal{F}(\mathcal{M})$, such that $Y \; R \; Y'$, and Y is in the A-connected component of X. Hence $\mathcal{M}, Y \models \varphi$, and by induction hypothesis, $\mathcal{M}', Y' \models \varphi$. The converse is similar. □

3.5.2 The equality negation task

We now now define the equality negation task for two processes as an action model. Recall that in the equality negation task, each process starts with an input value in the set $\{0, 1, 2\}$, and has to irrevocably decide on a value 0 or 1, such that the decisions of the two processes are the same if and only if their input values are different.

Let $\text{Ag} = \{b, w\}$ be the two agents (or processes), represented on the pictures in black and white, respectively. The atomic propositions are of the form input_a^i, for $a \in \text{Ag}$ and $i \in \{0, 1, 2\}$, meaning that agent a has input value i. The input model is $\mathcal{I} = \langle V_\mathcal{I}, S_\mathcal{I}, \chi_\mathcal{I}, \ell_\mathcal{I} \rangle$ where:

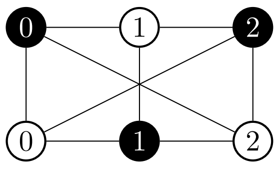
– The set of vertices of $\mathcal{I}$ is $V_\mathcal{I} = \text{Ag} \times \{0, 1, 2\}$

– The facets are of the form $\{(b, i), (w, j)\}$ for all i, j.

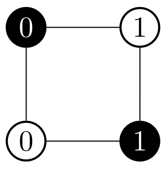
– The coloring $\chi_\mathcal{I} : V_\mathcal{I} \to \text{Ag}$ is the first projection $\chi_\mathcal{I}(a, i) = a$.

– The labeling of atomic propositions is $\ell_\mathcal{I}(a, i) = \{\mathsf{input}_a^i\}$.

The input model $\mathcal{I}$ is represented below. In the picture, a vertex $(a, i) \in V_\mathcal{I}$ is represented as a vertex of color a with value i.



$\mathcal{I}$

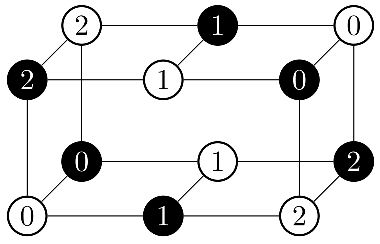


$G(\mathcal{T})$

We now define the action model $\mathcal{T} = \langle T, \sim, \mathsf{pre} \rangle$ that specifies the task. Since the only possible outputs are 0 and 1, there are four possible actions: $T = \{0, 1\}^2$, where by convention the first component is the decision of b, and the second component is the decision of w. Thus, two actions $(d_b, d_w) \sim_b (d'_b, d'_w)$ in T are indistinguishable by b when $d_b = d'_b$, and similarly for w. Finally, the precondition $\mathsf{pre}(d_b, d_w)$ specifies the task as expected: if $d_b = d_w$ then $\mathsf{pre}(d_b, d_w)$ is true exactly in the simplices of $\mathcal{I}$ which have different input values, and otherwise in all the simplices which have identical inputs. The associated simplicial action model $G(\mathcal{T})$ is depicted above, on the right.

The *output model* is obtained as the product update model $\mathcal{O} = \mathcal{I}[\mathcal{T}] = \langle V_\mathcal{O}, S_\mathcal{O}, \chi_\mathcal{O}, \ell_\mathcal{O} \rangle$. By definition, the vertices of $\mathcal{O}$ are of the form (a, i, E), where $(a, i) \in V_\mathcal{I}$ is a vertex of $\mathcal{I}$, and E is an equivalence class of $\sim_a$. But note that $\sim_a$ has only two equivalence classes, depending on the decision value (0 or 1) of process a. So, a vertex of $\mathcal{O}$ can be written as (a, i, d), for $d \in \{0, 1\}$, meaning intuitively that process a started with input i and decided value d. The facets of $\mathcal{O}$ are of the form $\{(b, i, d_b), (w, j, d_w)\}$ where either $i = j$ and $d_b \neq d_w$, or $i \neq j$ and $d_b = d_w$. The coloring $\chi_\mathcal{O}$ and labeling $\ell_\mathcal{O}$ behave the same as in $\mathcal{I}$.

The output model for the equality negation task is depicted below. The value written inside a node is the input value of the corresponding process. Decision values do not appear explicitly on the picture, but notice how the vertices are arranged as a rectangular cuboid: the vertices on the front face have decision value 0, and those on the rear face decide 1.



$\mathcal{O} = \mathcal{I}[\mathcal{T}]$

We would like to prove that this task is not solvable in the N layer message-passing model, i.e., that there is no morphism $\delta : \mathcal{I}[\mathcal{MP}_N] \to \mathcal{O}$ that makes the diagram of Definition 3.49 commute. So, we want to find an epistemic logic formula φ which is false in some world X of $\mathcal{I}[\mathcal{MP}_N]$, but true in its image $\delta(X)$. We now show that no such formula exists.

First, notice that there is a bisimulation between the input model $\mathcal{I}$ and the output model $\mathcal{O}$.

Lemma 3.58. *Let π denote the first projection simplicial map $\pi : \mathcal{O} \to \mathcal{I}$. Then, the relation R defined by $R := \{(\pi(X), X) \mid X \in \mathcal{F}(\mathcal{O})\} \subseteq \mathcal{F}(\mathcal{I}) \times \mathcal{F}(\mathcal{O})$ is a bisimulation between $\mathcal{I}$ and $\mathcal{O}$.*

Proof. The first condition of Definition 3.56 is trivially verified, because the labeling $\ell_{\mathcal{O}}$ is taken by definition from the first component. Let us check that condition (ii) is verified. Let X and X' be facets of $\mathcal{I}$ and $\mathcal{O}$ respectively, such that $X \ R \ X'$. Thus, we have $X = \{(b, i), (w, j)\}$ and $X' = \{(b, i, d_b), (w, j, d_w)\}$, for some i, j, d_b, d_w. Now let $a \in \mathrm{Ag}$ (w.l.o.g., let us pick $a = b$), and assume that there is some $Y \in \mathcal{F}(\mathcal{I})$ such that $b \in \chi(X \cap Y)$. So, Y can be written as $Y = \{(b, i), (w, j')\}$ for some j'. We now need to find a facet Y' of $\mathcal{O}$ that shares a b-colored vertex with X', and whose projection $\pi(Y')$ is Y. Thus, Y' should be of the form $Y' = \{(b, i, d_b), (w, j', d'_w)\}$, for some d'_w, such that $i = j' \iff d_b \neq d'_w$. But whatever the values of i, j', d_b are, we can always choose a suitable d'_w. This concludes the proof. The third condition (iii) is checked similarly. □

We can finally use Lemma 3.57 to show that no formula φ will allow us to prove the unsolvability of the equality negation task.

Lemma 3.59. *Let X be a facet of $\mathcal{I}[\mathcal{MP}_N]$ and let Y be a facet of $\mathcal{O}$ such that $\pi(X) = \pi(Y)$. Then for every positive formula φ we have the following: if $\mathcal{O}, Y \models \varphi$ then $\mathcal{I}[\mathcal{MP}_N], X \models \varphi$.*

Proof. Let φ be a positive formula and assume $\mathcal{O}, Y \models \varphi$. Since we have shown in Lemma 3.58 that $\pi(Y)$ and Y are bisimilar, by Lemma 3.57, we have $\mathcal{I}, \pi(Y) \models \varphi$. Since $\pi(Y) = \pi(X)$, this means that $\mathcal{I}, \pi(X) \models \varphi$, and by applying Lemma 3.15 we obtain $\mathcal{I}[\mathcal{MP}_N], X \models \varphi$. □

In the above lemma, the world Y should be thought of as a candidate for $\delta(X)$. The condition $\pi(X) = \pi(Y)$ comes from the commutative diagram of Definition 3.49. Thus, Lemma 3.59 says that we will never find a formula φ which is true in $\delta(X)$ but false in X.

Remark 3.60. Lemma 3.59 does not apply to consensus, since we know from Section 3.4.1 that there exists a formula proving its unsolvability. The reason is that the projection mapping $\pi : \mathcal{O} \to \mathcal{I}$ in consensus does not induce a bisimulation. More precisely, condition (ii) of Definition 3.56 does not hold. Indeed, if $X = \{(b, 0), (w, 1)\}$ and $X' = \{(b, 0, 1), (w, 1, 1)\}$ and $Y = \{(b, 0), (w, 0)\}$, then by definition of consensus there cannot exist a facet Y' with $Y \ R \ Y'$ and $b \in \chi'(X' \cap Y')$. Such a facet would have the form $Y' = \{(b, 0, 1), (w, 0, d)\}$, for a $d \in \{0, 1\}$, which is not a valid world in the output model of consensus for any decision d.

This reasoning can be adapted in the case of the k-set agreement task, to show that the input and output models are not bisimilar. Thus, the above argument cannot be used to show that there is no formula witnessing the impossibility of solving 2-set agreement. This remains an open question.

3.6 Extended DEL

In the usual DEL framework (see Section 3.1.3), when we start with some Kripke model M and apply an action model $\mathcal{A}$, we obtain a new Kripke model $M[\mathcal{A}]$. The worlds of $M[\mathcal{A}]$ are pairs (w, t), where w is a world of M and t is an action of $\mathcal{A}$. The intuitive meaning of the world (w, t) is: "I was in world w and then the action t occurred". By definition, the atomic propositions in the model $M[\mathcal{A}]$ are the same as in M: thus, we only consider formulas which say something about what the agents know about the original model M. Our logic does not allow us to write something like "I know that the action t occurred", even though this kind of consideration would make sense in the model $M[\mathcal{A}]$.

The idea of this section is simply to extend the set of atomic propositions $\widehat{\mathrm{At}} := \mathrm{At} \cup \{p_t \mid t \in T\}$, where the atom p_t means "the action t happened". In the language $\mathcal{L}_{CK}(\mathrm{Ag}, \widehat{\mathrm{At}})$, one can write epistemic logic formulas expressing what the agents know about the initial model, as well as what they know about the actions that occurred. To be able to interpret these formulas in the product-update model $M[\mathcal{A}]$, one must label the worlds of $M[\mathcal{A}]$ with the new atomic propositions: namely, the world (w, t) is given the label $L(w, t) := L(w) \cup \{p_t\}$. We will call this approach "Extended DEL". As noted by one of the referees of this thesis manuscript, it is actually a well-known construction in DEL. It is used for example in [90] to establish complexity results for the satisfiability of DEL formulas. More generally, the idea of allowing formulas talking about the actions that occur is related to extensions of DEL with history operators [111] or factual change [117]. Below, we describe the extended product-update model in terms of simplicial models.

Definition 3.61. Let $M = \langle V, S, \chi, \ell \rangle$ be a simplicial model and $\mathcal{A} = \langle T, \sim, \mathsf{pre} \rangle$ an action model. We write $\widehat{\mathrm{At}}$ for the *extended set of atomic propositions*, defined by $\widehat{\mathrm{At}} = \mathrm{At} \cup \{p_{a,E} \mid E \in T/\!\sim_a,\ a \in \mathrm{Ag}\}$, where $T/\sim_a$ denotes the set of all equivalence classes of $\sim_a$. Then, the (hybrid) *extended product-update simplicial model* $\widehat{M[\mathcal{A}]} = \langle V[\mathcal{A}], S[\mathcal{A}], \chi[\mathcal{A}], \widehat{\ell}[\mathcal{A}] \rangle$ is defined as follows. The three components $V[\mathcal{A}], S[\mathcal{A}], \chi[\mathcal{A}]$ are defined as in the usual product-update construction (Definition 3.31). In particular, the vertices $(v, E) \in V[\mathcal{A}]$ are pairs where $v \in V$ is a vertex of M and E is an equivalence class of $\sim_{\chi(v)}$. The enriched labeling $\widehat{\ell}[\mathcal{A}]$ maps each vertex $(v, E) \in V[\mathcal{A}]$ to the set of atomic propositions $\widehat{\ell}[\mathcal{A}](v, E) := \ell(v) \cup \{p_{\chi(v),E}\}$.

Remark 3.62. Note that the atomic propositions $p_{a,E}$ that we introduce are "local" propositions pertaining to one particular agent, as required by our definition of simplicial models. The fact that E is an equivalence class of $\sim_a$ is reminiscent of the construction of Section 3.2.4, where we make a non-local model local. Indeed, by doing this construction, we are in some sense "making the action model local". Here the equivalence class E has an intuitive meaning: the proposition $p_{a,E}$ means that "some action $t \in E$ occurred". The agent a necessarily knows this information, since E is an equivalence class of $\sim_a$.

In our case, we will use this extended product update in the following way. Given an input simplicial model $\mathcal{I}$ and a task action model $\mathcal{T}$, the *extended output model* $\widehat{\mathcal{O}}$ is defined as $\widehat{\mathcal{O}} = \widehat{\mathcal{I}[\mathcal{T}]}$. Recall from Definition 3.46 that the indistinguishability relation $\sim_a$ of a task action model $\mathcal{T}$ is defined as $t \sim_a t'$ if the agent a takes the same decision in t and t'. So, an equivalence class E of $\sim_a$ is a set of actions $t \in T$ where a takes the same decision value, say $d \in V^{out}$. Therefore, the new atomic propositions $p_{a,E}$ will be written decide_a^d, meaning that the agent a decides the output value d.

Consider the atomic propositions $\widehat{\mathrm{At}} = \{\mathsf{input}_a^i \mid a \in \mathrm{Ag}, i \in V^{in}\} \cup \{\mathsf{decide}_a^d \mid a \in \mathrm{Ag}, d \in V^{out}\}$. Formulas in $\mathcal{L}_{CK}(\mathrm{Ag}, \widehat{\mathrm{At}})$ can express what the agents know about the inputs of the other agents, but also what they know about their decisions. In this language, we will be able to write formulas witnessing the impossibility of solving equality negation and 2-set-agreement.

Suppose we have a protocol action model $\mathcal{A}$, and a candidate formula φ for proving the impossibility of solving the task $\mathcal{T}$ in $\mathcal{A}$. To be able to use Lemma 3.15 (as outlined in Section 3.3.3), we would also like the formula φ to make sense in the protocol model $\mathcal{I}[\mathcal{A}]$. However, the extended product-update $\widehat{\mathcal{I}[\mathcal{A}]}$ does not have information about decision values, but about the executions that occurred. In fact, it is precisely the role of the simplicial map $\delta : \mathcal{I}[\mathcal{A}] \to \mathcal{O}$ to assign decision values to each world of $\mathcal{I}[\mathcal{A}]$. Thus, given such a map δ, we can lift it to a map $\widetilde{\delta} : \widetilde{\mathcal{I}[\mathcal{A}]} \to \widehat{\mathcal{O}}$ as the following lemma states.

Lemma 3.63. *Let $M = \langle V, S, \chi, \ell \rangle$ be a simplicial model over the set of agents* Ag *and atomic propositions* At*, and let $\delta : M \to \mathcal{O}$ be a morphism of simplicial models. Then there is a unique model of the form $\widetilde{M} = \langle V, S, \chi, \widetilde{\ell} \rangle$ over $\widehat{\mathrm{At}}$, with the same underlying simplicial complex as M, and where $\widetilde{\ell}$ agrees with ℓ on* At*, such that $\widetilde{\delta} : \widetilde{M} \to \widehat{\mathcal{O}}$ is still a morphism of simplicial models.*

Proof. All we have to do is label the worlds of M with the decide_a^d atomic propositions, so that the map $\widetilde{\delta}$ is a morphism of simplicial models. Thus, we define $\widetilde{\ell} : V \to \mathscr{P}(\widehat{\mathrm{At}})$ as $\widetilde{\ell}(v) = \ell(v) \cup \{\mathsf{decide}_a^d\}$, where $a = \chi(v)$ and $\mathsf{decide}_a^d \in \ell_{\widehat{\mathcal{O}}}(\delta(v))$. Then $\widetilde{\delta}$ is still a chromatic simplicial map (since we did not change the underlying complexes nor their colors), and moreover we have $\widetilde{\ell}(v) = \ell_{\widehat{\mathcal{O}}}(\delta(v))$ for all v. The model $\widetilde{M}$ is unique since any other choice of $\widetilde{\ell}(v)$ would have broken this last condition, so δ would not be a morphism of simplicial models. □

3.6.1 Unsolvability of equality negation and 2-set-agreement

We apply this "Extended DEL" framework to prove the two impossibility results where the standard method did not succeed: equality negation and 2-set-agreement.

Equality negation

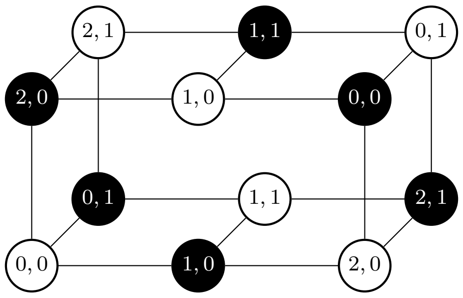
We prove the impossibility of solving equality negation in the N-layer message-passing model $\mathcal{MP}_N$. To do so, we rely on the following formula, written in the language $\mathcal{L}_{CK}(\mathrm{Ag}, \widehat{\mathrm{At}})$:

$$\varphi \;=\; \bigwedge_{a,i,d} \mathsf{input}_a^i \wedge \mathsf{decide}_a^d \implies \Big((\mathsf{input}_{\overline{a}}^i \wedge \mathsf{decide}_{\overline{a}}^{\overline{d}}) \vee (\mathsf{input}_{\overline{a}}^{\overline{i}} \wedge \mathsf{decide}_{\overline{a}}^{d}) \Big)$$

where $\overline{a}$, $\overline{i}$, $\overline{d}$ denote values different from a, i, d, respectively. Note that $\overline{a}$ and $\overline{d}$ are uniquely defined (since there are only two agents and two decision values), but for $\overline{i}$, there are two possible inputs different from i. So, for example, $\mathsf{input}_a^{\overline{0}}$ is actually a shortcut for $\mathsf{input}_a^1 \vee \mathsf{input}_a^2$.

This formula simply expresses the specification of the task: if process a has input i and decides d, then the other process should either have the same input and decide differently, or have a different input and decide the same. Then hopefully φ would be true in every world of the output model, but would fail somewhere in the protocol model $\mathcal{I}[\mathcal{MP}_N]$, meaning that the N-layer message-passing model is not powerful enough to obtain this knowledge.

The extended output model $\widehat{\mathcal{O}}$ for equality negation is represented below. It is almost the same as $\mathcal{O}$, except that the decision values are now explicit. A vertex represented with color a and value (i, d) means that it is labeled with the atomic propositions input_a^i and decide_a^d. It is easily checked that the formula φ is true in every world of $\widehat{\mathcal{O}}$.



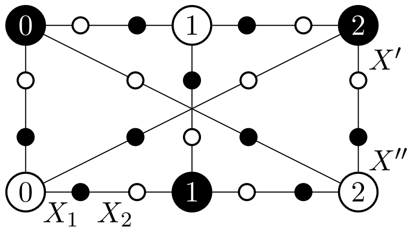
We can finally prove that the equality negation task is not solvable:

Theorem 3.64. *Equality negation for two processes is not solvable in the N-layer message-passing model.*

Proof. Let us assume by contradiction that the task is solvable, i.e., by Definition 3.49, there exists a morphism of simplicial models $\delta : \mathcal{I}[\mathcal{MP}_N] \to \mathcal{O}$ that makes the diagram commute. By Lemma 3.63, we can lift δ to a morphism $\widetilde{\delta} : \widetilde{\mathcal{I}[\mathcal{MP}_N]} \to \widehat{\mathcal{O}}$ between the extended models. As we remarked earlier, the formula φ is true in every world of $\widehat{\mathcal{O}}$. Therefore, it also has to be true in every world of $\widetilde{\mathcal{I}[\mathcal{MP}_N]}$. Indeed, for any world X, since $\widehat{\mathcal{O}}, \widetilde{\delta}(X) \models \varphi$, and $\widetilde{\delta}$ is a morphism, by Lemma 3.15, we must have $\widetilde{\mathcal{I}[\mathcal{MP}_N]}, X \models \varphi$. We will now derive a contradiction from this fact.

$$\begin{tikzcd}
 & \mathcal{I}[\mathcal{MP}_N] \arrow[dl, "\widetilde{\pi}"'] \arrow[d, "\delta"] & \widetilde{\mathcal{I}[\mathcal{MP}_N]} \arrow[d, "\widetilde{\delta}"] \\
\mathcal{I} & \mathcal{O} \arrow[l, "\pi"] & \widehat{\mathcal{O}}
\end{tikzcd}$$

Recall that the protocol model $\mathcal{I}[\mathcal{MP}_N]$ is just a subdivision of the input model $\mathcal{I}$, as depicted below for $N = 1$. (For simplicity, some input values have been omitted in the vertices on a subdivided edge; it is the same input as the extremity of the edge which has the same color. Also, the picture shows only one subdivision, but our reasoning is unrestricted and it applies to any number of layers N.)



Let us start in some world $_1$ on the $(w, 0) - (b, 1)$ edge. In the world X_1, the two processes have different inputs. Since in $\widetilde{\mathcal{I}[\mathcal{MP}_N]}$, the formula φ is true in X_1, the decision values have to be the same. Without loss of generality, let us assume that in X_1, both processes decide 0. We then look at the next world X_2, which shares a black vertex with X_1. Since the inputs are still 0 and 1, and φ is true, and we assumed that process b decides 0, then the white vertex of X_2 also has to decide 0.

We iterate this reasoning along the $(w, 0) - (b, 1)$ edge, then along the $(b, 1) - (w, 2)$ edge, and along the $(w, 0) - (b, 2)$ edge: all the vertices on these edges must have the same decision value 0. Thus, on the picture, the top right $(b, 2)$ corner has to decide 0, as well as the bottom right $(w, 2)$ corner.

Now in the world X', the two input values are equal, so the processes should decide differently. Since the black vertex decides 0, the white vertex must have decision value 1. If we keep going along the rightmost edge, the decision values must alternate: all the black vertices must decide 0, and the white ones decide 1. Finally, we reach the world X'', where both decision values are 0, whereas the inputs are both 2. So the formula φ is false in X'', which is a contradiction. $\square$

2-set agreement

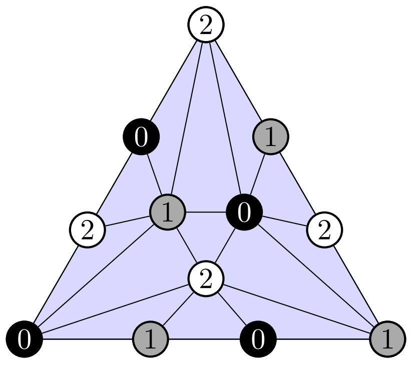
We now use the same reasoning to prove the impossibility of solving 2-set agreement (Section 3.4.3). To express the property that must be achieved in order to solve the task, we take the following formula φ, which expresses the specification of the 2-set-agreement task:

$$\begin{aligned}\varphi \quad = \quad & \forall d_0, d_1, d_2.\ \mathsf{decide}_b^{d_0} \wedge \mathsf{decide}_g^{d_1} \wedge \mathsf{decide}_w^{d_2} \\ & \qquad \Longrightarrow \quad |\{d_0, d_1, d_2\}| \leq 2 \ \wedge\ \exists a.\ \mathsf{input}_a^{d_0} \ \wedge\ \exists a.\ \mathsf{input}_a^{d_1} \ \wedge\ \exists a.\ \mathsf{input}_a^{d_2}\end{aligned}$$

The quantifiers $\forall$ and $\exists$ are just shortcuts for conjunctions and disjunctions ranging over all possible values for d_0, d_1, d_2 and a (there are only finitely many of them). The condition $|\{d_0, d_1, d_2\}| \leq 2$ cannot be expressed as it is, but we could simply replace it by "true" or "false" in each case, depending on the values of d_0, d_1, d_2. It is easily checked that the formula φ is true in every world of $\widehat{\mathcal{O}}$. We can finally prove that 2-set agreement is not solvable by the immediate snapshot protocol.

Proof of Theorem 3.55. Assume by contradiction that the task is solvable, i.e., that there exists a morphism $\delta : \mathcal{I}[\mathcal{IS}] \to \mathcal{I}[\mathcal{T}]$ that makes the diagram of Definition 3.49 commute. By Lemma 3.63, δ can be lifted to a map $\widetilde{\delta} : \widetilde{\mathcal{I}[\mathcal{IS}]} \to \widehat{\mathcal{I}[\mathcal{T}]}$. Our goal is to contradict Lemma 3.15, using the formula φ defined in the beginning of this section. As we remarked earlier, φ is true in every world of $\widehat{\mathcal{I}[\mathcal{T}]}$. Thus, all we have to do is prove that there exists one world X of $\widetilde{\mathcal{I}[\mathcal{IS}]}$ where the formula is false.

Consider one facet of the input model $\mathcal{I}$ where all the agents start with distinct values, for example, $I_{012} = \{(b,0),(g,1),(w,2)\} \in \mathcal{I}$. This facet induces a subcomplex of $\mathcal{I}[\mathcal{IS}]$, which we write $I_{012}[\mathcal{IS}]$. It consists of all the worlds of the form (I_{012}, c^{012}), where c is a sequential partition of the agents. The figure below represents the subcomplex $I_{012}[\mathcal{IS}]$; the values written inside the nodes represent the input_a^i atomic propositions.



Since it is a chromatic subdivision, it is known that $I_{012}[\mathcal{IS}]$ is a pseudomanifold with boundary [64]. Moreover, in the extended model $\widetilde{\mathcal{I}[\mathcal{IS}]}$, the decision values induce a Sperner coloring on its boundary. Indeed, let us first look at the extremal vertices of the triangle, for example the bottom left $(b,0)$ vertex. The view of this vertex is $0\bot\bot$, so, it also belongs to the subcomplex $I_{000}[\mathcal{IS}]$ where everyone started with value 0. In this subcomplex, all the decision values must be 0, otherwise the formula φ would be false (which would conclude our proof). So, this vertex must have the decision value 0. Simularly, the bottom right vertex $(g,1)$ must decide 1, and the top vertex $(w,2)$ must decide 2. We can reason similarly with the edge vertices. For example, the view of the vertices on the bottom edge is $01\bot$. So, they also belong to the subcomplex $I_{010}[\mathcal{IS}]$ where white has value 0. In this subcomplex, all the decision values must be either 0 or 1, otherwise the formula φ would not be satisfied.

Thus, by Sperner's Lemma, there must be one world X of $I_{012}[\mathcal{IS}]$ with three different decision values. Then, it is easily checked that $\widetilde{\mathcal{I}[\mathcal{IS}]}, X \not\models \varphi$, which concludes the proof. $\square$

3.6.2 Perspectives of the Extended DEL approach

It is interesting to compare the epistemic formulas that we used in Section 3.6.1 to prove the unsolvability of equality negation and 2-set-agreement, with the ones that we used in Sections 3.4.1 and 3.4.2 for consensus and approximate agreement. In the case of consensus and approximate agreement, we did not need the "Extended DEL" framework. The formula φ was only talking about the knowledge that the agents acquired about the input values. These two formulas are quite informative about the underlying task: they tell us that the main goal of the consensus task is to achieve common knowledge, while the goal of approximate agreement is to reach some finite depth of nested knowledge operators.

On the other hand, the formulas for equality negation and 2-set-agreement are less informative: they simply state the specifications of the tasks. They do not even seem to be talking about knowledge, since there are no K or C operators in the formulas. In fact, their epistemic content is hidden in the decide_a^d atomic propositions. Indeed, their semantics in $\widetilde{\mathcal{IA}}$ is referring to the decision map δ, which assigns a decision value d to each vertex of $\mathcal{I}[\mathcal{A}]$. The fact that we assign decisions to vertices means that each process must decide its output *solely according to its knowledge*. So, it seems that the use of Extended DEL is making the epistemic content of the formulas implicit. This goes against the main motivation of our DEL approach, which is to explain the topological impossibility proofs in terms of knowledge.

Despite the fact that it produces less informative formulas, the "Extended DEL" proof method has two benefits. The first one is that it is actually able to prove *any* impossibility result. Indeed, let $\mathcal{T} = \langle T, \sim, \mathsf{pre}\rangle$ be a task action model, on the input model $\mathcal{I}$, and let $\mathcal{A}$ be a protocol action model. Remember that the elements of T are functions $t : \mathrm{Ag} \to V^{out}$ assigning a decision value to each agent. Let φ denote the following formula:

$$\varphi = \bigwedge_{X \in \mathcal{F}(\mathcal{I})} \left(\bigwedge_{a \in \mathrm{Ag}} \mathsf{input}_a^{X(a)} \implies \bigvee_{\substack{t \in T \\ \mathcal{I}, X \models \mathsf{pre}(t)}} \bigwedge_{a \in \mathrm{Ag}} \mathsf{decide}_a^{t(a)} \right) \tag{3.2}$$

where $X(a)$ denotes the input value of process a in the input simplex X. Then we get the following theorem.

Theorem 3.65. *The task $\mathcal{T}$ is solvable in the protocol $\mathcal{A}$ if and only if there exists an extension $\widetilde{\mathcal{I}[\mathcal{A}]}$ of $\mathcal{I}[\mathcal{A}]$ (assigning a single decision value to each vertex of $\mathcal{I}[\mathcal{A}]$) such that φ from (3.2) is true in every world of $\widetilde{\mathcal{I}[\mathcal{A}]}$.*

Proof. ($\Rightarrow$): We already proved this direction in Section 3.6.1. Assume that the task is solvable, i.e., by Definition 3.49, there is a morphism $\delta : \mathcal{I}[\mathcal{A}] \to \mathcal{I}[\mathcal{T}]$ such that $\pi \circ \delta = \pi$. The model $\widetilde{\mathcal{I}[\mathcal{A}]}$ is given Lemma 3.63, where the assignment of decision values is the one given by δ. Then φ is easily seen to be true in every world of $\widetilde{\mathcal{I}[\mathcal{T}]}$, and by Lemma 3.15, it is also true in every world of $\widetilde{\mathcal{I}[\mathcal{A}]}$.

($\Leftarrow$): For the converse, assume that there is a model $\widetilde{\mathcal{I}[\mathcal{A}]}$ where the formula φ is true in every world. Then we build a map $\delta : \mathcal{I}[\mathcal{A}] \to \mathcal{I}[\mathcal{T}]$ as follows. If a vertex (i, E) of $\widetilde{\mathcal{I}[\mathcal{A}]}$, colored by agent a, is labeled with the atomic proposition decide_a^d, we send it to the vertex $\delta(i, E) := (i, d)$ of $\mathcal{I}[\mathcal{T}]$.

By definition, we have the commutative diagram $\pi(i,E) = i = \pi \circ \delta(i,E)$. We now need to show that δ is a morphism of simplicial models. The coloring and labeling maps are preserved, since by definition they just copy the coloring and labeling of $\mathcal{I}$. We still have to prove that δ sends simplices to simplices. Let Y be a facet of $\mathcal{I}[\mathcal{A}]$. Let $X = \pi(Y) \in \mathcal{F}(\mathcal{I})$ be the facet of the input complex corresponding to the initial values of the processes in the execution Y. Then we have $\mathcal{I}[\mathcal{A}], Y \models \bigwedge_{a \in \mathrm{Ag}} \mathsf{input}_a^{X(a)}$, and since $\widetilde{\mathcal{I}[\mathcal{A}]}, Y \models \varphi$, by modus ponens there must be some action $t \in T$, with $\mathcal{I}, X \models \mathsf{pre}(t)$, such that $\widetilde{\mathcal{I}[\mathcal{A}]}, Y \models \bigwedge_{a \in \mathrm{Ag}} \mathsf{decide}_a^{t(a)}$. Thus, the vertex v of Y which is colored by a must be labeled with the atomic proposition $\mathsf{decide}_a^{t(a)}$, and so the map δ sends it to the vertex $(i, t(a))$ of $\mathcal{I}[\mathcal{T}]$. By definition of the action model, since $\mathcal{I}, X \models \mathsf{pre}(t)$, the set of vertices $\delta(Y) = \{(i, t(a)) \mid i \in X, a = \chi(i)\}$ is a simplex of $\mathcal{I}[\mathcal{T}]$. Therefore, the map δ is a simplicial map. □

This theorem implies that the situation of Section 3.5 cannot happen with the "Extended DEL" approach: if the task is not solvable, there necessarily exists a world X of $\widetilde{\mathcal{I}[\mathcal{A}]}$ where the formula fails. Of course, finding such a world is usually the hard part of an impossibility proof, but at least we know it exists. In fact, in the particular case of read/write protocols (or, equivalently, layered message-passing or immediate-snapshot), the solvability of tasks is known to be undecidable when there are more than three processes [43, 67]. Thus, according to Theorem 3.65, given a formula φ, the problem of deciding whether there exists a number of layers N and an extension of $\mathcal{I}[\mathcal{MP}_N]$ which validates a given formula φ, is also undecidable.

The second benefit of the "Extended DEL" framework is that it gives us a way of using epistemic logic directly as a specification language for tasks. Indeed, notice that in Theorem 3.65, we characterized the solvability of a task without referring to $\mathcal{T}$ itself: the formula φ contains all the information of $\mathcal{T}$. Thus, instead of relying on the commutative diagram of Definition 3.49, we can specify a task directly as a logical formula. Of course, we need not restrict ourselves to formulas such as φ which are direct translations of the input/output relation specifying a task. One could decide to pick a more informative formula, with an interesting epistemic content, and study whether this amount of knowledge can be reached in some computational model.

3.7 Conclusion and future work

The main contribution of this chapter is given in Theorem 3.20, which is the equivalence between Kripke models and simplicial models. This theorem establishes a bridge between the field of epistemic logic and distributed computability. It has important consequences for both fields. From the point of view of epistemic logic, this equivalence shows that Kripke models actually contain hidden topological information, which can be uncovered by the use of simplicial models. For distributed computing, it allows us to interpret epistemic logic formulas on the chromatic simplicial complexes that appear in impossibility proofs. The proof method outlined in Section 3.3.3 gives a more concrete meaning to the abstract topological arguments for unsolvability: in order to solve a task, the processes must acquire some amount of knowledge.

We have used our DEL framework to study the solvability of four tasks: consensus, approximate agreement, 2-set agreement and equality negation. The first two examples are quite encouraging, since the epistemic logic formulas that we use in the impossibility proofs shed a new light on the conditions that must be attained in order to solve these tasks. For the other two examples, we have not been able to find interesting formulas explaining the unsolvability of the tasks. The use of the "Extended DEL" framework allows us to conclude the proofs, but it comes with a major drawback: the epistemic content of the formulas is hidden in the decide_a^d atomic propositions.

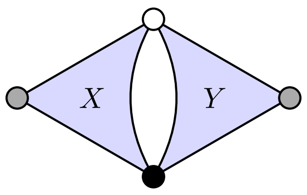
Perhaps another way to understand these two tasks would be to extend the language of formulas in a different way. Instead of adding new atomic propositions as we did in the Extended DEL approach, it would be interesting to introduce new epistemic operators, besides K_a and C_A. Indeed, the solvability of the k-set agreement task is known to be related to the higher-dimensional connectedness properties of the protocol complex. It would be very interesting to find a formalization of such topological properties in terms of knowledge, and thus obtain a generalization from common knowledge (that is tightly related to 1-dimensional connectedness) to other forms of knowledge (whose semantics would be related to higher-dimensional connectedness).

Future work. Many interesting questions are left for future work. We have developed here all our theory on pure simplicial complexes, where all the facets are of the same dimension. This means that the number of agents is the same in every world. In contrast, the systems that are studies in fault-tolerant distributed computing often have a varying number of processes (agents), since processes may crash during the execution. To model such systems, we would simply need to consider simplicial models where the underlying complex is not pure. As far as we know, this kind of situation with a variable number of agents is not usually considered in standard epistemic logic. Indeed, the usual Kripke models cannot express which agent are present or not in a given world. The simplicial setting looks like an elegant way of modeling these situations.

Another restriction that we have, which is very related to the previous one, is that the truth of a formula $M, X \models \varphi$ is defined only when X is a *facet* of M. It would be interesting to define what it means to interpret a formula in a lower-dimensional simplex $Y \subsetneq X$. This would allow us to model *sub-executions* where some processes do not participate in the computation. Indeed, our current DEL framework only allows us to work with computational models without crashes (see the discussion in Section 1.1.3). From the point of view of epistemic logic, this seems related to notions of *group knowledge* and *distributed knowledge*.

An important notion of epistemic logic that we have not considered is *belief*. Belief behaves a lot like knowledge, except that an agent may believe some formula φ which is false. After discussions with Hans van Ditmarsch, it seems that belief can be expressed elegantly in simplicial models, by equipping them with an idempotent map from vertices to vertices. More generally, there are many interesting multi-agent logics besides $\mathbf{S5_n}$, which are modeled by Kripke frames whose $\sim_a$ relations are not equivalence relations. Finding simplicial versions of these kinds of frames is an interesting open question.

Finally, another variation of our simplicial models that we could study, is one relying not on simplicial complexes but on *simplicial sets*. Simplicial sets are more general than simplicial complexes, since they allow things like self-loops, or having several edges between the same pair of vertices, and so on in higher dimensions. In particular, this would allow us to represent Kripke frames which are not proper: in a simplicial complex, we cannot have two triangles with the same three vertices, but in a simplicial set this is allowed. In fact, this would allow us to model much more than that. For instance, we could have the simplicial set:



Here, the agents white and black, individually, cannot distinguish between the two worlds X and Y. But if we combine their knowledge, then their *distributed knowledge* contains enough information to distinguish the two worlds. We do not know of a concrete example where such a situation has been studied, neither from the field of epistemic logic, nor distributed computing.

CHAPTER 4

Towards directed topological semantics

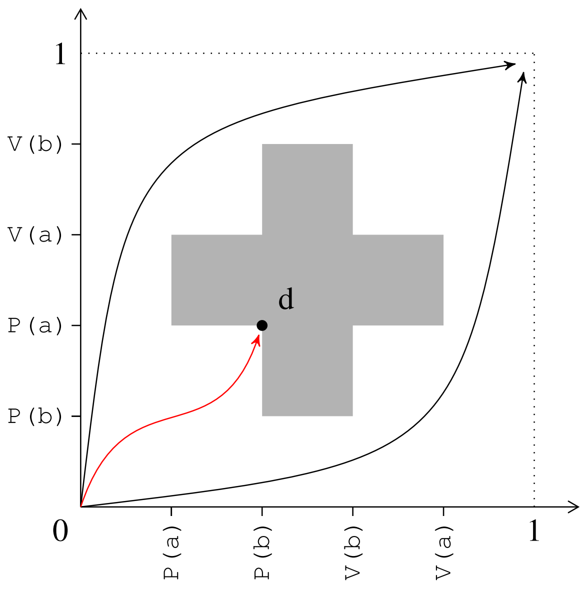
The topological methods for studying fault-tolerant protocols that we presented in Chapter 1 were discovered and developed in the beginning of the 1990's. Roughly at the same period, another kind of geometric model for concurrent programs was developed [105, 118], with close connections to *directed algebraic topology* [49]. Since then, this approach has yielded many fascinating results, both in mathematics and computer science. On the theoretical side, it helped develop the theory of directed homotopy and homology [39, 48, 108, 56, 28]. More concrete applications include deadlock detection [37] and state-space reduction [35] for concurrent programs. A recent book sums up most of the important advances that have been made in this field [36].

While the directed space models and the simplicial complex models both constitute algebraic topological semantics for concurrent programs, they make a very different use of topology, and their goals are very different. Indeed, on the one hand, the chromatic simplicial complexes of Chapter 1 model the local information that each process acquires during an execution (i.e., its *knowledge*, c.f. Chapter 3). On the other hand, the directed space semantics is concerned with studying the different ways in which a program can go from its initial state to its final state. Thus, it is typically used to study objects such as locks and semaphores, which alter the flow of the computation. This is in contrast with the wait-free objects that we have been studying in the previous chapters. However, it is in fact possible to use directed topology to study wait-free protocols; this approach has started to be studied in [55].

Directed algebraic topology is based on *directed topological spaces*, that is, topological spaces equipped with a notion of *direction*. There are many variants of this notion: continuous ones, such as partially-ordered spaces [30, 98], local po-spaces [38], streams [81] or d-spaces [56]; as well as combinatorial ones such as pre-cubical sets or, as they are called in computer science, Higher Dimensional Automata (HDA) [105]. We will only use HDAs in this chapter, but as a first intuitive introduction to the ideas of directed topology, we use continuous pictures to illustrate our explanations. Consider the following protocol for two processes P_0 and P_1, which is using two locks a and b. Using Dijkstra's classical notation for semaphores, we write $P(x)$ to acquire the lock x, and $V(x)$ to release it.

$$\begin{aligned} P_0 : &\quad P(a); P(b); V(b); V(a) \\ P_0 : &\quad P(b); P(a); V(a); V(b) \end{aligned}$$

So, process P_0 first tries to take the lock a, then the lock b, and then it releases both locks in reverse order. For process P_1, the role of a and b is switched. This protocol is nicknamed the "Swiss cross", referring to the picture below, representing the directed space associated with this protocol. It consists of the two-dimensional square $[0,1] \times [0,1]$, where the gray area, which is called the *forbidden region*, has been removed. Each axis is labeled with actions from one process: the x-axis corresponds to process P_0, while the y-axis corresponds to process P_1. The points in the gray area would correspond to states where both processes are holding the same lock at the same time, which is forbidden.



An execution of the protocol corresponds to a *path* in this space, starting at the origin $(0,0)$, and *increasing* in both components. Three examples of paths are depicted. Intuitively, when a path goes right, process P_0 is making progress, and when it goes up, process P_1 is making progress. Paths that are not increasing do not make sense from a computational point of view, since processes cannot go backwards in their program. Thus, this topological space is *directed* in the sense that paths can only go in one direction. The point labeled d is a *deadlock*: when a path reaches this point, it cannot make any more progress, and will never be able to reach the endpoint $(1,1)$. In other words, the protocol will never terminate. The central idea of this approach is that paths should be considered only up to *di-homotopy*, that is, up to continuous deformation. In the example above, even though there are six possible interleavings of the actions of the processes, there are only three maximal paths up to di-homotopy. This reflects the idea that some actions commute, that is, the order in which they occur does not affect the computation.

This chapter presents some preliminary results relating these directed topological models with the simplicial complex approach for fault-tolerant distributed computing. More precisely, we focus on a combinatorial version of these directed spaces, *Higher-Dimensional Automata*. On these structures, various notions of paths can be defined, *cube chains* [119] and *carrier sequences* [34]. We show how these two notions are related, respectively, to the set-sequential traces and the interval traces that we mentioned in Section 2.2.3 in the context of set-linearizability and interval-linearizability. In the case of cube chains and set-sequential traces, we go a bit further and show how we can recover the standard chromatic subdivision by considering *partial cube chains* (see Theorem 4.24). These results are first steps towards a better understanding of the relationship between the protocol complex and the directed space associated to a given protocol. At the end of the chapter, we discuss future research directions that could lead to interesting results relating the two fields. Eventually, we would like to answer the following questions:

- Can we reconstruct the protocol complex from the directed semantics of the objects that we use?
- Can we derive impossibility results relying only on the topological properties of the directed space?

As we explain below, the answer to these two questions as they currently stand is no: the directed space lacks some information regarding the *local views* of the processes. Hence, the real question is: what is the right way to extend the directed space semantics in order to model the notion of view?

Limitations of the directed space semantics. As a counter-example to the questions above, we exhibit two concurrent objects that have the same directed semantics, but not the same power in terms of task solvability. We fix a set of 3 processes, $\{P, Q, R\}$.

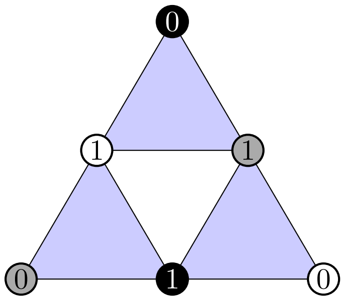
- *Test-and-set:* at the beginning the memory cell has value 0. The test-and-set$()$ method atomically sets it to 1, and returns the old value.
- *Compare-and-swap:* at the beginning the memory cell has value $\perp$. The compare-and-set(x, y) method atomically compares the current value of the flag to x, and, if they are equal, it stores the value y in the memory cell. In both cases, the old value of the flag is returned.

It is well known in distributed computing [63] that the test-and-set object has consensus number 2 while the compare-and-swap object has consensus number ∞. For three processes, this means that test-and-set cannot solve consensus, but compare-and-swap can. We consider the following two protocols using these objects.

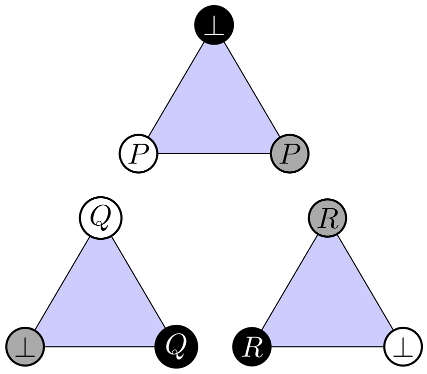
(1) There is one test-and-set object, t. Each process calls the method t.test-and-set$()$ exactly once. Thus, one process ("the winner") receives value 0, and the two others receive value 1.

(2) There is one compare-and-set object, c. Process X calls the method c.compare-and-swap$(\perp, X)$ exactly once. Thus, one process ("the winner") receives value $\perp$, and the two others receive the name of the winner.

The corresponding protocol complexes are represented below, where the processes P, Q, R are represented in black, gray, white, respectively.

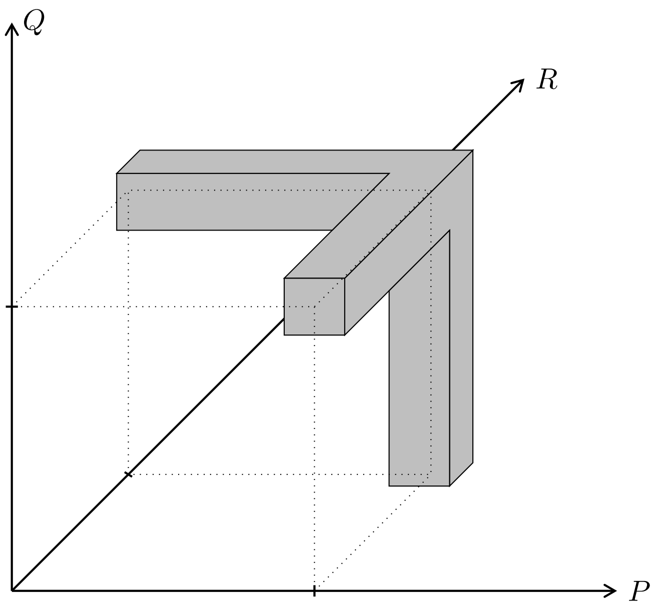


Protocol complex for (1)



Protocol complex for (2)

However, both these protocols are associated with the same directed space. Indeed, there is a conflict regarding which process goes first, but afterwards the other two processes can go concurrently. The corresponding directed space is depicted below, where, the gray region is the forbidden region. There are three di-homotopy classes of paths in this space: one for each face by which we can exit the dotted cube.



As we mentioned before, the missing information in this directed space is the notion of *view*. The di-homotopy classes of paths indicate all the possible executions that can happen, that is, they correspond to the facets of the protocol complex. However, it does not tell us how these facets are "glued together" in the protocol complex. Indeed, from the point of view of a given process two executions might be indistinguishable, even if from a global perspective they are not equivalent.

Related work. A first paper by Goubault, Mimram and Tasson [55] studied the relationship between the directed semantics of the write-snapshot object (called scan-update in their paper), and the associated protocol complex. As in the examples above, a bijection between the di-homotopy classes of paths and the facets of the protocol complex is easily achieved. The difficulty lies in recovering the gluing information between those facets. To do so, they explore various notions of view: one based on execution traces (which was the starting point of our work in Section 2.1), and one based on interval orders.

The main result of the chapter, Theorem 4.24, is very much inspired from the paper by Ziemiański [119] about cube chains. In this paper, the cube chains are used to provide a combinatorial description (more precisely, a CW-complex) of the space of directed paths in the geometric realization of an HDA. The other important notion of path on HDAs that we use, carrier sequences, was invented by Fajstrup in [34]. The bijection between carrier sequences and interval orders was also independently discovered by Fahrenberg et al. in an unpublished work about pomset languages for HDAs [31].

Plan. In Section 4.1, we start by recalling the important definitions that we use in this chapter: HDAs and the various notions of paths on HDAs. Then, we prove our main theorem in Section 4.2. Finally, in Section 4.3 we discuss the future research directions on this topic, and the potential applications of the results we already have.

4.1 Preliminaries

This section is not intended as a general introduction to the directed semantics approach for concurrency. We simply provide the technical definitions that we will be working with in the rest of the chapter. We refer the reader to [36] for an explanation of how these notions are used in the context of concurrent programming. In particular, we will not discuss how to associate an HDA with a given program, nor how to define the geometric realization of an HDA as a d-space.

4.1.1 Higher-dimensional automata

As in the previous chapters, we write $[n]$ for the set $\{0,\dots,n\}$ of cardinality $n+1$.

Definition 4.1. A *pre-cubical set* K is a sequence of disjoints sets $(K_n)_{n\geq 0}$, equipped with face maps $\partial_i^\varepsilon : K_{n+1}\to K_n$, for $\varepsilon\in\{-,+\}$ and $i\in[n]$, satisfying the following pre-cubical relations for $i<j$:

$$\partial_i^\varepsilon\circ\partial_j^\eta = \partial_{j-1}^\eta\circ\partial_i^\varepsilon$$

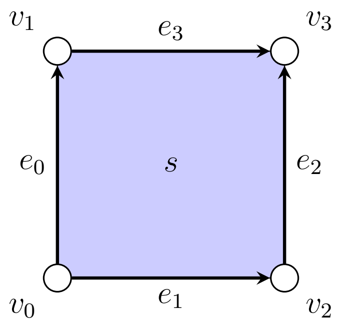
Elements of K_0 are called *vertices*, elements of K_1 are called *edges*, elements of K_2 are called *squares*. More generally, elements of K_n are called n*-cubes*.

We denote by $\partial^- : K_n\to K_0$ the n-fold composition of maps of the form ∂_i^- (the indexes do not matter according to the pre-cubical relations). The map $\partial^+ : K_n\to K_0$ is defined similarly. When K is a pre-cubical set, we write $c\in K$ to denote that $c\in K_n$ for some n. Given two cubes $c,c'\in K$, if $c'=\partial_{i_0}^+\circ\cdots\circ\partial_{i_k}^+(c)$ for some suitable indexes $i_0,\dots,i_k$, $k\geq 0$, we say that c' is a *front face* of c and we write $c\rhd^+ c'$. Similarly, if $c'=\partial_{i_0}^-\circ\cdots\circ\partial_{i_k}^-(c)$ for some $i_0,\dots,i_k$, we say that c' is a *back face* of c and we write $c\rhd^- c'$. Note that with this definition, a cube is neither a front face nor a back face of itself. A pre-cubical set is of *dimension* n if $K_n\neq\varnothing$ and $K_k=\varnothing$ for all $k>n$.

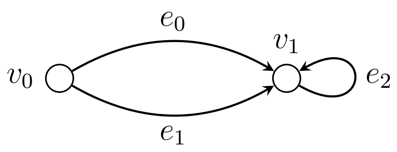
Example 4.2. Below is depicted a pre-cubical complex of dimension 2. It has four vertices $K_0 = \{v_0,v_1,v_2,v_3\}$, four edges $K_1=\{e_0,e_1,e_2,e_3\}$ and one square $K_2=\{s\}$. The face maps are defined as follows:

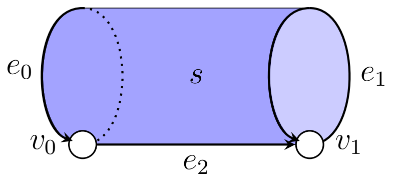
$$\begin{array}{cccccc}
\partial_0^-(e_0)=v_0 & \partial_0^-(e_1)=v_0 & \partial_0^-(e_2)=v_2 & \partial_0^-(e_3)=v_1 & \partial_0^-(s)=e_0 & \partial_1^-(s)=e_1 \\
\partial_0^+(e_0)=v_1 & \partial_0^+(e_1)=v_2 & \partial_0^+(e_2)=v_3 & \partial_0^+(e_3)=v_3 & \partial_0^+(s)=e_2 & \partial_1^+(s)=e_3
\end{array}$$

Notice that the pre-cubical relations are satisfied, for instance, $v_1=\partial_0^-\circ\partial_1^+(s)=\partial_0^+\circ\partial_0^-(s)$. The meaning of those relations can be understood on the picture below: intuitively, they say that starting from s, projecting first horizontally and then vertically, or vice versa, gives the same result. Moreover, one can check that s has three front faces (e_2, e_3 and v_3) and three back faces (e_0, e_1 and v_0). Note that v_1 and v_2 are neither front nor back faces of s. Finally, we have $\partial^-(s)=v_0$ and $\partial^+(s)=v_3$.



Example 4.3. Pre-cubical sets of dimension 1 correspond to *directed graphs*, where K_0 is the set of vertices, K_1 the set of edges, and $\partial_0^-,\partial_0^+ : K_1\to K_0$ are respectively the *source* and *target* maps. Notice however that it is a rather unrestricted notion of graphs: they are allowed to contain self-loops, as well as parallel edges. The same phenomenon occurs with higher-dimensional pre-cubical sets. For instance, the cylinder depicted below on the right has one square s, three edges e_0, e_1, e_2 and two vertices v_0, v_1. The face maps of the square are defined as $\partial_0^-(s) = e_0$ and $\partial_0^+(s) = e_1$ and $\partial_1^+(s) = \partial_1^-(s) = e_2$. Thus, the square s is rolled on itself to form a cylinder, since it has the same vertical source and target, e_2.



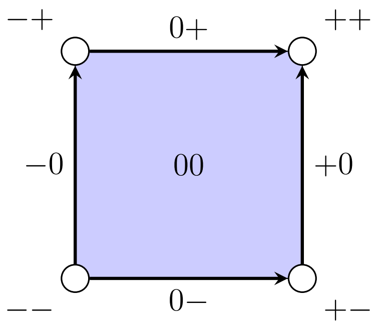


The pre-cubical set of Example 4.2 is the 2-dimensional version of what we call the *standard n-cube*. It can be defined combinatorially in any dimension $n \geq 0$ as follows.

Definition 4.4. We define a pre-cubical set $\square^n$ called the *standard n-cube* as follows.

- $\square_k^n \subseteq \{-,+,0\}^n$ is the set of sequences of length n on the set $\{-,+,0\}$ where the symbol 0 occurs exactly k times. In particular, $\square_k^n = \varnothing$ for $k > n$.
- The face map $\partial_i^\varepsilon : \square_{k+1}^n \to \square_k^n$ replaces the i-th occurrence of 0 by ε (numbering starts at 0).

Example 4.5. For $n = 2$, the pre-cubical set $\square^2$ is indeed the square of Example 4.2.



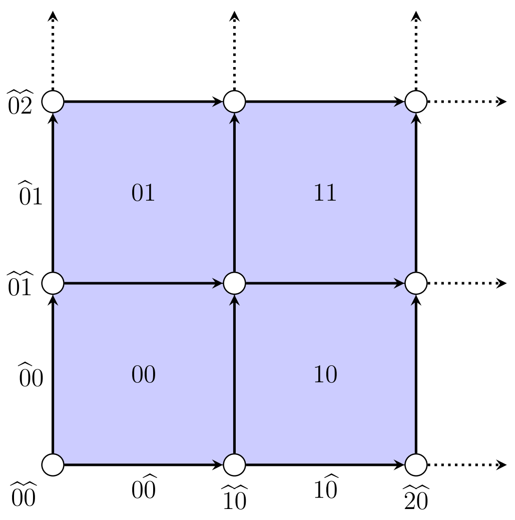
The computational interpretation of the standard n-cube is the following: n asynchronous processes are about to execute one operation each. The computation starts at the source vertex $\mathsf{s} = - - \cdots -$, and ends at the target vertex $\mathbf{t} = + + \cdots +$. Going from '$-$' to '$+$' in the i-th dimension means that process i has performed its operation. We will also be interested in interpreting situations where each process performs more that one operation. The corresponding pre-cubical set is defined as follows.

Definition 4.6. We define the n-dimensional *stack of cubes* $\boxplus^n$ as follows.

- $\boxplus_k^n \subseteq (\mathbb{N} \cup \widehat{\mathbb{N}})^n$, where $\widehat{\mathbb{N}} = \{\widehat{n} \mid n \in \mathbb{N}\}$, is the set of sequences of length n that contain exactly k symbols in $\mathbb{N}$. In particular, $\boxplus_k^n = \varnothing$ for $k > n$.
- The face map $\partial_i^- : \boxplus_{k+1}^n \to \boxplus_k^n$ replaces the i-th symbol in $\mathbb{N}$, say $n \in \mathbb{N}$, by $\widehat{n} \in \widehat{N}$.
- The face map $\partial_i^+ : \boxplus_{k+1}^n \to \boxplus_k^n$ replaces the i-th symbol in $\mathbb{N}$, say $n \in \mathbb{N}$, by $\widehat{n+1} \in \widehat{N}$.

Note that this pre-cubical set has infinitely many cubes.

Example 4.7. Parts of the 2-dimensional stack of cubes $\boxplus^2$ is represented below. The full pre-cubical set extends infinitely to the right and up.



Since we are interested in modeling *wait-free* computations, we will usually consider executions where each process executes a bounded number of operations. If each process executes exactly k operations, we will consider paths in $\boxplus^n$ which start at the source vertex $\mathbf{s} = \widehat{0} \cdots \widehat{0}$ and end at the target vertex $\mathbf{t} = \widehat{k} \cdots \widehat{k}$. So, we only use a finite portion of $\boxplus^n$. A pre-cubical set equipped with *source* and *target* vertices is usually called a *Higher-Dimensional Automaton* (HDA).

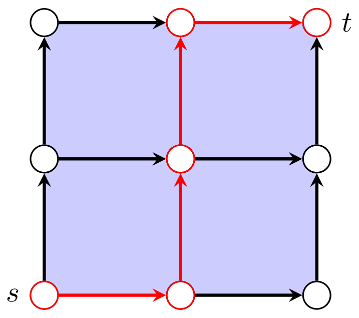
Definition 4.8. An *HDA* is a triple (K, s, t) where K is a pre-cubical set, the vertex $s \in K_0$ is the *source* and the vertex $t \in K_0$ is the *target*.

Remark 4.9. Alternatively, the source can also be called the *initial state*, and the target is a *final state*. When interpreting concrete programs, one often considers a set $F \subseteq K_0$ of final states; but here we are only interested in studying the space of paths between two given points. Usually, some topological conditions are imposed on HDAs in order to avoid pathological cases: in [31], they are required to be *non-selflinked* and *geometric*, while in [119], they must have a *proper non-looping length covering*. Here, we only work on $\square^n$ and on finite subsets of $\boxplus^n$, both of which are very non-pathological since they can be embedded on a standard N-cube (possibly for $N > n$). Of course, it would be interesting to identify more precisely the class of pre-cubical sets on which our constructions make sense. Another difference with the usual definition of HDAs is that they are usually equipped with a labeling of the edges. We do not include these labels here since they do not play any role in the theorems that we want to prove. When using HDAs to model actual programs, the labels would be added to indicate which operations the processes are performing.

4.1.2 Notions of paths on HDAs

There are several ways to define a combinatorial notion of *path* on an HDA. The most simple one is to consider sequences of consecutive edges from an initial state to a final state, as one would do on a graph. We will call those *edge paths*. However, this does not make use of the higher-dimensional topological information in HDAs, and thus it is usually not a relevant notion of path in this context. Other notions of paths have been defined that provide a combinatorial counterpart to the space of directed paths in the geometric realization of an HDA: *cube chains* in [119] and *carrier sequences* in [34].

Edge paths. Let (K, s, t) be an HDA. An *edge path* from s to t in K is a finite sequence of edges $e_1, \ldots, e_k \in K_1$ such that $\partial_0^-(e_1) = s$, $\partial_0^+(e_k) = t$ and for all $1 \leq i < k$, $\partial_0^+(e_i) = \partial_0^-(e_{i+1})$. Note that this is simply a graph-theoretic path on the 1-skeleton of K, that is, on the graph whose set of vertices is K_0, whose set of edges is K_1, and whose source and target maps are ∂_0^- and ∂_0^+. The picture below depicts a path (in red) from $s = \widehat{0}\widehat{0}$ to $t = \widehat{2}\widehat{2}$ in $\boxplus^2$.



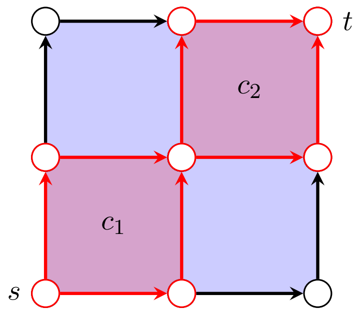
Cube chains. Cube chains were introduced in [119] in order to provide a minimal combinatorial model of the space of directed paths on the geometric realization of a pre-cubical set. The main difference between edge paths and cube chains is that instead of a sequence of edges, we may have a sequence of cubes of any dimension.

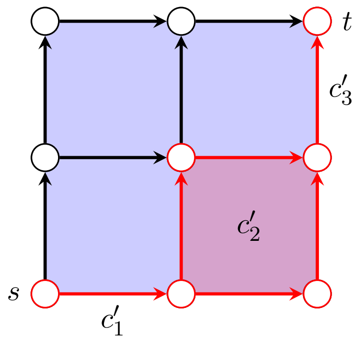
Definition 4.10. Let (K, s, t) be an HDA. A *cube chain* from s to t in K is a sequence of cubes $\overline{\mathbf{c}} = (c_1, \ldots, c_\ell)$ where $c_i \in K_{n_i}$ $(n_i > 0)$ such that:

- $\partial^-(c_1) = s$,
- $\partial^+(c_\ell) = t$,
- and for all $1 \leq i < \ell$, $\partial^+(c_i) = \partial^-(c_{i+1})$.

Recall that the maps ∂^- and ∂^+ (without index) denote the iterated composition of the face maps ∂_j^- and ∂_j^+, so that $\partial^-(c)$ and $\partial^+(c)$ are always vertices. So, two consecutive cubes are always linked by their extremal vertices. The sequence $(n_1, \ldots, n_\ell)$ is called the *type* of the cube chain $\overline{\mathbf{c}}$. The *dimension* of $\overline{\mathbf{c}}$ is $\dim(\overline{\mathbf{c}}) = n_1 + \ldots + n_\ell - \ell$; and $n_1 + \ldots + n_\ell$ is the *length* of $\overline{\mathbf{c}}$. The vertices s and t are called respectively the *source* and *target* of $\overline{\mathbf{c}}$. The set of cube chains from s to t in K is written $\mathbf{Ch}(K)_s^t$.

Two cube chains from $s = \widehat{0}\widehat{0}$ to $t = \widehat{2}\widehat{2}$ in $\boxplus^2$ are represented below (in red). The one on the left consists of two cubes c_1, c_2, both of which are of dimension 2. So, its type is $(2, 2)$; its dimension is 2, and its length is 4. The one on the right consists of three cubes, c_1', c_2', c_3'. Its type is $(1, 2, 1)$, its dimension is 1 and its length is 4.





Remark 4.11. Note that an edge path is a particular case of a cube chain, where all cubes are actually edges (i.e., of dimension 1). More precisely, the edge paths are exactly the cube chains of dimension 0. Notice that the term "dimension" does not correspond to the intuitive notion of edges, squares, cubes and so on. Rather, it measures how much concurrency has occurred in the execution represented by a path. The terminology comes from [119], where it actually corresponds to the dimension of the cells of the CW-complex that they construct using cube chains.

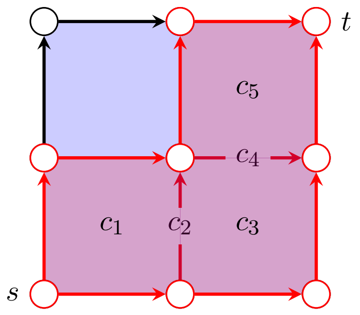
Carrier sequences. The third notion of paths is the most permissive. It was introduced in [34] and later used in [31] to provide a notion of language for HDAs. In a cube chain, two consecutive cubes are always linked by a vertex, which is the target of one cube and the source of the next one. The idea behind carrier sequences is that we are allowed to link two consecutive cubes by any other cube (of lower dimension) that is a front face of one cube, and a back face of the next one.

Definition 4.12. Let (K, s, t) be an HDA. A *carrier sequence* from s to t in K is a sequence of cubes $\bar{\mathbf{c}} = (c_1, \ldots, c_\ell)$ such that $s \lhd^- c_1 \rhd^+ c_2 \lhd^- c_3 \rhd^+ \cdots \lhd^- c_\ell \rhd^+ t$. In other words:

- s is a back face of c_1 (since s is a vertex, this means that $\partial^-(c_1) = s$);
- for all $1 \leq i \leq \lfloor \frac{\ell}{2} \rfloor$, c_{2i} is a front face of c_{2i-1} and a back face of c_{2i+1}; and
- t is a front face of c_ℓ, i.e., $\partial^+(c_\ell) = t$.

Note that, necessarily, the number ℓ must be odd. Intuitively, the even numbered cubes c_{2i} are the "interfaces" by which the two cubes c_{2i-1} and c_{2i+1} are linked together. The set of carrier sequences from s to t in K is written $\mathbf{Car}(K)_s^t$.

The picture below represents a carrier sequence from $s = \widehat{0}\widehat{0}$ to $t = \widehat{2}\widehat{2}$ in $\boxplus^2$. It is composed of five cubes c_1, c_2, c_3, c_4, c_5, where c_1, c_3, c_5 are squares and c_2, c_4 are edges. One can check that we indeed have $s \lhd^- c_1 \rhd^+ c_2 \lhd^- c_3 \rhd^+ c_4 \lhd^- c_5 \rhd^+ t$.



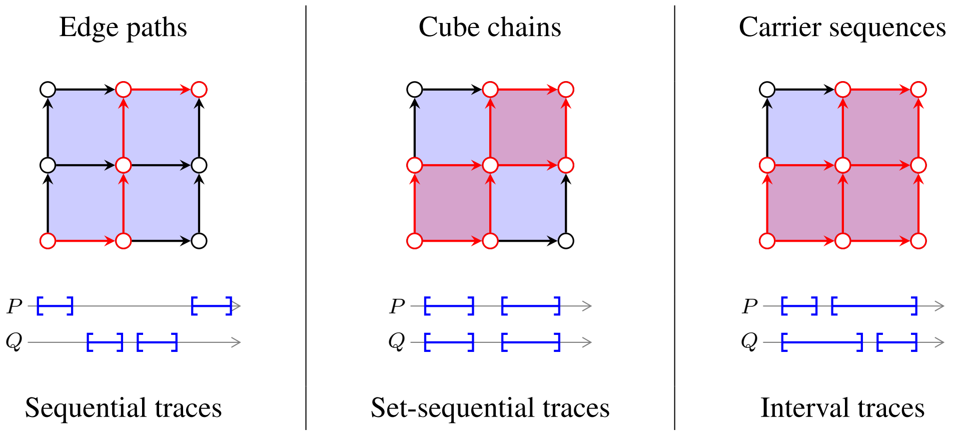
Remark 4.13. Cube chains (and, by Remark 4.11, edge paths too) are particular cases of carrier sequences. Indeed, they correspond exactly to the carrier sequences where every even-numbered cube c_{2i} is a vertex.

4.2 Relating trace semantics and paths on HDAs

The usual computational interpretation of the directed space semantics for concurrency is that directed paths correspond to executions of a program. As we mentioned before, the pre-cubical set $\boxplus^n$ represents the situation where n asynchronous processes each execute a sequence of operations. Specifically, a path from $s = \widehat{0} \cdots \widehat{0}$ to $t = \widehat{k} \cdots \widehat{k}$ should represent an execution where each process performs a sequence of exactly k operations. In this section, we explore a bit further the different notions of paths and their computational interpretation in terms of execution traces.

4.2.1 Three simple bijections

First, we relate the three notions of paths of Section 4.1.2 (edge paths, cube chains and carrier sequences) to the three variants of linearizability that we studied in Section 2.2.3 (linearizability [71], set-linearizability [99] and interval-linearizability [18]). Recall that the main purpose of those linearizability-based techniques is to produce a concurrent specification, by providing a (usually simpler) specification, which only specifies the behavior of the object on traces of a specific *shape*. Namely, standard linearizability starts with a specification that deals with *sequential traces*; set-linearizability specifies the *set-sequential traces*; and interval-linearizability specifies *interval-sequential traces*, that is, traces of any shape.



The picture above sums up the three results that we will prove in the remainder of the section. Namely, edge paths correspond to sequential traces; cube chains correspond to set-sequential traces; and carrier sequences correspond to general traces. A little subtlety is that in order to obtain a bijection, we must consider traces up to commutations of invocations and commutations of responses (this was discussed in the last paragraph of Section 2.2.2). A bit more abstractly, this means that the notion that we really capture is Lamport's so-called *precedence partial order* [85] between operations. This is the approach that has been taken in [31], which is based on partial orders rather than execution traces.

Edge paths and sequential traces

This correspondence is well known: edge paths correspond to *interleavings* of the operations of the processes. Interleaving semantics are known to be well-suited for studying concurrent objects in which every action appears to happen atomically. One of the reasons for the introduction of HDAs was to get away from interleaving semantics in order to model *true concurrency*, where operations can happen at the same time. In order to prove this result formally, we recall the definition of a sequential trace. Since we only care about the "shape" of the traces and not about the operations, we do not include input and output values.

Consider n processes and let $\{0, \dots, n-1\}$ be the set of process numbers. A *trace* is a finite word on the alphabet $\mathcal{A} = \{\mathsf{i}_i, \mathsf{r}_i \mid 0 \leq i < n\}$ that is *alternating*, that is, that begins with an invocation and alternates between invocations and responses (see Section 2.2.2). We write $\mathcal{T}$ for the set of all traces. A trace is *sequential* if every invocation i_i is immediately followed by a matching response r_i. For $k \in \mathbb{N}$, we write Seq_k for the set of sequential traces where each process performs exactly k operations, that is, traces that contain exactly k occurrences of i_i and r_i for each $0 \leq i < n$. We also write $\mathbf{Edg}(K)_s^t$ for the set of edge paths from s to t in K. We also write $\widehat{\mathbf{0}}$ for the vertex $\widehat{0} \cdots \widehat{0}$ of $\boxplus^n$, and $\widehat{\mathbf{k}}$ for the vertex $\widehat{k} \cdots \widehat{k}$.

Theorem 4.14. *There is a bijection between* Seq_k *and* $\mathbf{Edg}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$.

Proof. A sequential trace $T \in \mathsf{Seq}_k$ can be seen as a sequence of integers $i_1, \ldots, i_{kn} \in \{0, \ldots, n-1\}$, where each process number j appears exactly k times in s. Indeed, since each invocation is immediately followed by a matching response, we can group the two symbols together and write i instead of $\mathsf{i}_i \cdot \mathsf{r}_i$.

Let $p \in \mathbf{Edg}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$ be an edge path from $\widehat{\mathbf{0}}$ to $\widehat{\mathbf{k}}$ in $\boxplus^n$. An induction on k shows that it must be of length kn. So, p is given by a sequence of edges $e_1, \ldots, e_{kn} \in \boxplus_1^n$. By definition, an edge of $\boxplus^n$ is an n-tuple of elements of $\mathbb{N} \cup \widehat{\mathbb{N}}$, where exactly one component is in $\mathbb{N}$. If the j-th component (numbering starts at 0) of e is in $\mathbb{N}$, we associate the process number j with it. This gives us a sequence of process numbers $i_1, \ldots, i_{kn}$, that is, a sequential trace. Moreover, each process number $0 \leq j < n$ must appear exactly k times in the sequence. Indeed, we can prove by induction on the length of the path that an edge path in $\boxplus^n$ which starts at $\widehat{\mathbf{0}}$, and whose associated sequence contains a occurrences of j, ends at a vertex whose j-th component is $\widehat{a}$.

Conversely, given a sequential trace, viewed as a sequence of integers $i_1, \ldots, i_{kn}$, we can construct an edge path $e_1, \ldots, e_{kn}$ as follows. For an index $1 \leq r \leq kn$ and a process number j, we write $\#_r(j) := |\{s \mid i_s = j, 1 \leq s < r\}|$ for the number of occurrences of j in the sequence $i_1, \ldots, i_{r-1}$. Then, the edge e_r for $1 \leq r \leq kn$ has the symbol $\#_r(i_r)$ in its i_r-th component, and the symbol $\widehat{\#_r(j)}$ in its j-th component for $j \neq i_r$. This is indeed an edge path: indeed, consider two consecutive edges e_r and e_{r+1}. We have $\#_{r+1}(i_r) = \#_r(i_r) + 1$, and for $j \neq i_r$, $\#_{r+1}(j) = \#_r(j)$. So all the components are unchanged, except for the i_r-th one which has been incremented by one. So, we have $\partial_0^+(e_r) = \partial_0^-(e_{r+1})$ since the effect of ∂_0^+ is to increment the component without a hat, that is, the i_r-th one. Similarly, a simple calculation shows that $\partial_0^-(e_1) = \widehat{\mathbf{0}}$ and $\partial_0^+(e_{kn}) = \widehat{\mathbf{k}}$.

Checking that the composition $\mathsf{Seq}_k \to \mathbf{Edg}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}} \to \mathsf{Seq}_k$ is the identity is straightforward. For the other direction, let $e_1, \ldots, e_{kn}$ be an edge path, and $i_1, \ldots, i_{kn}$ its associated sequence. A simple induction on r shows that the value (with or without hat) of the j-th component of e_r is $\#_r(j)$. Since the construction also preserves the position of the hats, we obtain the same edge path that we started with. $\square$

Cube chains and set-sequential traces

Set-sequential traces are a somewhat ad-hoc formalism that came up in order to specify useful concurrent objects such as set-agreement objects and immediate-snapshot. In contrast, cube chains are a canonical in the sense that they provide a combinatorial model of the space of directed paths on a pre-cubical set, which is *minimal* among such constructions. Thus, this relationship between cube chains and set-sequential traces is surprising, and might provide some insight on why the immediate-snapshot object is so important in distributed computing.

Recall that a trace $T \in \mathcal{T}$ is *set-sequential* if whenever a response occurs, all the pending invocations must receive a response before a new invocation can occur. More formally, for every prefix of T of the form $T' \cdot \mathsf{r}_i \cdot \mathsf{i}_j$, there must be no pending invocation in $T' \cdot \mathsf{r}_i$ (i.e., for all j, it contains the same number of i_j and r_j symbols). We write SetSeq_k for the set of set-sequential traces that contain exactly k invocations i_i and k responses r_i for each process i. As we mentioned earlier, we will actually consider these traces up to commutations of invocations and of responses. As in Section 2.2.2, we write $T \equiv T'$ to denote that T' is obtained from T by reordering the blocks of consecutive invocations or consecutive responses.

Theorem 4.15. *There is a bijection between* $\mathsf{SetSeq}_k/\equiv$ *and* $\mathbf{Ch}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$.

Proof. First, remark that we have a simple representation of the elements of $\mathsf{SetSeq}_k/\equiv$. They are sequences of sets of process numbers, $I_1, \dots, I_m$ where each $I_s \subseteq \{0, \dots, n-1\}$, such that each $0 \leq j < n$ appears in exactly k of those sets. Indeed, such a sequence can be turned into a set-sequential trace by replacing each set $I = \{i_1, \dots, i_r\}$ by the word $\mathsf{i}_{i_1} \cdots \mathsf{i}_{i_r} \cdot \mathsf{r}_{i_1} \cdots \mathsf{r}_{i_r}$, and concatenating them. Conversely, we can associate with a set-sequential trace the sequence of sets of participating processes; this is well-defined on the quotient set since it is invariant under reordering of consecutive invocations and consecutive responses. It is straightforward to check that composition in both ways gives the identity.

The remainder of the proof is quite similar to the one of Theorem 4.14. Recall that the cubes of $\boxplus^n$ of dimension d are n-tuples of elements of $\mathbb{N} \cup \widehat{\mathbb{N}}$, with exactly d components in $\mathbb{N}$. We write $c_r[j]$ for the j-th component of the cube c_r. Consider a cube chain $\overline{\mathbf{c}} = c_1, \dots, c_m \in \mathbf{Ch}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$. For each c_r, we define $I_r = \{j \mid c_r[j] \in \mathbb{N}\}$ the set of indexes at which c_r has a symbol in $\mathbb{N}$. This gives us a sequence of sets of process numbers, i.e., an equivalence class of set-sequential traces. To check that it is an element of $\mathsf{SetSeq}_k/\equiv$, we must prove that each process number $j \in \{0, \dots, n-1\}$ appears exactly k times in the sequence $I_1, \dots, I_m$. As in the previous proof, we can show by induction that a cube chain in $\boxplus^n$ which starts at $\widehat{\mathbf{0}}$, and whose associated sequence contains a occurrences of j, ends at a vertex whose j-th component is $\widehat{a}$. So, in order to end at $\widehat{\mathbf{k}}$, every j must appear k times.

The converse is also very similar to the one of the previous theorem. Assume given a sequence of sets $I_1, \dots, I_m$, where every process number $0 \leq j < n$ appears exactly k times. Given an index $1 \leq r \leq m$ and a process number j, we write $\#_r(j) := |\{s \mid j \in I_s, 1 \leq s < r\}|$ for the number of occurrences of j in $I_1, \dots, I_{r-1}$. Then, we can construct a cube chain $c_1, \dots, c_m \in \mathbf{Ch}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$ as follows. The cube c_r, of dimension $|I_r|$, is obtained by putting the symbol $\#_r(j)$ in the j-th component if $j \in I_r$, and $\widehat{\#_r(j)}$ in the j-th component if $j \notin I_r$. Checking that this is still a cube chain is a straightforward calculation as in the case of edge paths: two consecutive cubes c_r and c_{r+1} have the same values in all components, except for the j-th ones with $j \in I_r$, which have been incremented in c_{r+1}. Since they are also the components without hats in c_r, we have $\partial^+(c_r) = \partial^-(c_{r+1})$. The conditions on $\widehat{\mathbf{0}}$ and $\widehat{\mathbf{k}}$ are checked similarly.

For convenience, we name $f : \mathbf{Ch}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}} \to \mathsf{SetSeq}_k/\equiv$ and $g : \mathsf{SetSeq}_k/\equiv \to \mathbf{Ch}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$ the two maps defined above. To show that $f \circ g$ is the identity, consider a sequence $s = I_1, \dots, I_m$. Its image by g is $g(s) = c_1, \dots, c_m$, where each cube c_r has symbols without hats (in $\mathbb{N}$) exactly at the indexes $j \in I_r$. Hence $f \circ g(s) = I_1, \dots, I_m$ by definition of f. Conversely, take a cube chain $\overline{\mathbf{c}} = c_1, \dots, c_m$ from $\widehat{\mathbf{0}}$ to $\widehat{\mathbf{k}}$ in $\boxplus^n$, and let $f(\overline{\mathbf{c}}) = I_1, \dots, I_m$ be its image by f. We prove the following by induction on r: for all $0 \leq j < n$, $c_r[j] = \#_r(j)$ (we disregard the hats for now). For $r = 1$, $\#_r(j) = 0$ for all j by definition, and since we must have $\partial^-(c_1) = \widehat{\mathbf{0}}$, we also have $c_1[j] = 0$ or $\widehat{0}$ for all j. Now assume by induction hypothesis that some r, $c_r[j] = \#_r(j)$ for all j. As we said before, $\#_{r+1}(j) = \#_r(j) + 1$ if $j \in I_r$ and $\#_{r+1}(j) = \#_r(j)$ otherwise. By definition of ∂^+, the vertex $v = \partial^+(c_r)$ has value $\widehat{c_r[j] + 1} = \widehat{\#_{r+1}(j)}$ in the j-th component if $j \in I_r$, and $\widehat{c_r[j]} = \widehat{\#_{r+1}(j)}$ otherwise. Since we must also have $v = \partial^-(c_{r+1})$, the j-th component of c_{r+1} must have value $\#_{r+1}(j)$ (with or without hat) for all j. What we have shown so far proves that the cubes of $g \circ f(\overline{\mathbf{c}})$ have the same values as those of $\overline{\mathbf{c}}$. We still need to check that the hats are at the same places, but this is obvious, since the set I_r is by definition the set of indexes j such that $c_r[j] \in \mathbb{N}$, and the function g also puts values in $\mathbb{N}$ for the indexes in I_r, and values in $\widehat{\mathbb{N}}$ otherwise. $\square$

Carrier sequences and interval traces

The third correspondence is between carrier sequences and general (unconstrained) traces, which are also called "interval-sequential" in [18]. As this name suggests, traces are closely related to so-called *interval orders*, via Lamport's precedence partial order that we mentioned in Chapter 2, just after Definition 2.24. The correspondence between carrier sequences and interval orders has been noticed independently by Fahrenberg et al. [31].

Like in the previous case, we want to consider traces only up to commutations of invocations and of responses. In fact, this is already how it is done in [18], where a trace is represented as a sequence $I_1, R_1, \dots, I_m, R_m$ where each I_r (resp., R_r) is a non-empty set of invocations (resp., responses). In our case, the only information contained in invocations and responses is the process number, so we can represent an equivalence class of traces as a sequence $I_1, R_1, \dots, I_m, R_m$ where each $I_r, R_r \subseteq \{0, \dots, n-1\}$ are non-empty sets of process numbers. Moreover, recall that traces are always assumed to be alternating, which can be formulated as follows when working with equivalence classes. The sequence $I_1, R_1, \dots, I_m, R_m$ is *alternating* if:

(i) each process begins with an invocation: if $j \in R_r$ then there exists $s \leq r$ such that $j \in I_s$,

(ii) no two consecutive invocations: if $j \in I_r$ and $j \in I_s$ for $r < s$, then $j \in R_t$ for some $r \leq t < s$,

(iii) no two consecutive responses: if $j \in R_r$ and $j \in R_s$ for $r < s$, then $j \in I_t$ for some $r < t \leq s$.

Constructing the bijection between $\mathcal{T}/\equiv$ and these sequences is quite straightforward. Given a trace, we obtain the corresponding sequence by replacing each maximal block of invocations (resp., responses) by the set I_r (resp., R_r) of process numbers that appear in it. This is well-defined on $\mathcal{T}/\equiv$ since it does not depend on the choice of the representative of an equivalence class. Conversely, given a sequence of sets, we can reconstruct a trace by choosing an arbitrary ordering of the invocations/responses in each set. It is easy to see that composing these two maps gives the identity in both directions.

Remark 4.16. To be completely formal, we should also show that, along this bijection, the above definition of "alternating" coincides with the one on regular traces. Although it is not difficult to prove, it is quite cumbersome to write fully formally. Since it is not our main point of interest here, we leave out the details.

An alternating sequence of sets is *complete* if moreover it has the following property:

(iv) each process ends with a response: if $j \in I_r$ then there exists $s \geq r$ such that $j \in R_s$.

We write $\mathcal{T}_k$ for the set of traces that contain exactly k occurrences of the symbol i_i, and k occurrences of r_i, for each process number $0 \leq i < n$. Thus, an equivalence class in $\mathcal{T}_k/\equiv$ can be seen as a sequence $I_1, R_1, \dots, I_m, R_m$ that is alternating, complete, and where each process i appears in exactly k of the sets R_r (automatically, the same is true for invocations).

Theorem 4.17. *There is a bijection between $\mathcal{T}_k/\equiv$ and $\mathbf{Car}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$.*

Proof. We define a function $f : \mathbf{Car}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}} \to \mathcal{T}_k/\equiv$ as follows. Let $c_1, \dots, c_{2m-1}$ be a carrier sequence from $\widehat{\mathbf{0}}$ to $\widehat{\mathbf{k}}$, i.e., such that $\widehat{\mathbf{0}} \lhd^- c_1 \rhd^+ c_2 \lhd^- c_3 \rhd^+ \cdots \lhd^- c_{2m-1} \rhd^+ \widehat{\mathbf{k}}$. For each cube c_r, we define $S_r = \{j \mid c_r[j] \in \mathbb{N}\}$ the set of indexes at which c_r has a symbol in $\mathbb{N}$. We also take $S_0 = S_{2m} = \varnothing$, which amounts to the same construction on the cubes $c_0 = \widehat{\mathbf{0}}$ and $c_{2m} = \widehat{\mathbf{k}}$. Then, for each $1 \leq \ell \leq m$ we choose $I_\ell = S_{2\ell-1} \setminus S_{2\ell-2}$ and $R_\ell = S_{2\ell-1} \setminus S_{2\ell}$. This gives us a sequence of sets $I_1, R_1, \dots, I_m, R_m$. We now prove that this sequence is alternating. First, note that whenever $c_r \lhd^+ c_s$, then $S_r \subseteq S_s$, and the same is true when $c_r \lhd^- c_s$. That is because the face maps ∂_i^+ and ∂_i^- always replace symbols in $\mathbb{N}$ by symbols in $\widehat{\mathbb{N}}$. So, we have $S_0 \subseteq S_1 \supseteq S_2 \subseteq \cdots \supseteq S_{2m-1} \subseteq S_{2m}$.

(i) if $j \in R_\ell$, then $j \in S_{2\ell-1}$. Take the largest $r < 2\ell - 1$ such that $j \notin S_r$ (worst case, $r = 0$ works). Then r must be even because of the inclusions above, and therefore $j \in I_{\frac{r}{2}+1} = S_{r+1} \setminus S_r$.

(ii) Assume $j \in I_\ell$ and $j \in I_t$ for $\ell < t$. Then in particular, $j \in S_{2\ell-1}$ and $j \notin S_{2t-2}$. Take the largest index r with $2\ell - 1 \leq r < 2t - 2$, such that $j \in S_r$. It exists since $2\ell - 1$ works, and it must be odd because of the inclusions. Then, we get $j \in R_{\lfloor \frac{r}{2} \rfloor + 1} = S_r \setminus S_{r+1}$.

(iii) Assume $j \in R_\ell$ and $j \in R_t$ for $\ell < t$. Then in particular, $j \notin S_{2\ell}$ and $j \in S_{2t-1}$. Take the largest index r with $2\ell \leq r < 2t - 1$, such that $j \notin S_r$. It exists since 2ℓ works, and it must be even because of the inclusions. Then, we get $j \in I_{\frac{r}{2}+1} = S_{r+1} \setminus S_r$.

It is also complete:

(iv) if $j \in I_\ell$, then $j \in S_{2\ell-1}$. Take the smallest $r > 2\ell - 1$ such that $j \notin S_r$ (it exists since $r = 2m$ works). Then r must be even, and we get $j \in R_{\frac{r}{2}} = S_{r-1} \setminus S_r$.

Moreover, each process number j appears exactly k times in the sets $R_1, \ldots, R_m$. To show this, we prove the following property by induction on $0 \leq \ell \leq m$: if j appears a times in $R_1, \ldots, R_\ell$, then the cube $c_{2\ell}$ has value a or $\widehat{a}$ at position j. Therefore, since $c_{2m} = \widehat{\mathbf{k}}$ has value $\widehat{k}$ at position j, the only possibility is to have k occurrences of j in $R_1, \ldots, R_m$. The property is true for $\ell = 0$. Assume by induction hypothesis that it is true for ℓ, and let a be the number of occurrences of j in $R_1, \ldots, R_{\ell+1}$. We distinguish two cases, depending whether or not $j \in R_{\ell+1}$.

– If $j \in R_{\ell+1}$, by induction hypothesis the cube $c_{2\ell}$ has value $a - 1$ (or $\widehat{a-1}$) in its j-th component. Since $c_{2\ell} \lhd^- c_{2\ell+1}$, the cube $c_{2\ell+1}$ also has the same value, because the maps ∂_i^- do not change the value of the components. Moreover, $j \in R_{\ell+1}$ implies that $j \in S_{2\ell+1}$ and $j \notin S_{2\ell+2}$. This means that the j-th component of $c_{2\ell+1}$ is in $\mathbb{N}$, while the j-th component of $c_{2\ell+2}$ is in $\widehat{\mathbb{N}}$. Thus, one of the face maps ∂_i^+ modified the j-th component, which is now $\widehat{a}$ in $c_{2\ell+2}$.

– If $j \notin R_{\ell+1}$, by induction hypothesis the cube $c_{2\ell}$ has value a or $\widehat{a}$ in its j-th component. As in the previous case, the same is true for $c_{2\ell+1}$. Then, because $j \notin R_{\ell+1}$, the j-th component of $c_{2\ell+1}$ and $c_{2\ell+1}$ either both have a hat, or neither of them do. In both cases, that means the face maps did not affect the j-th component, which is still a or $\widehat{a}$.

Therefore, the sequence $I_1, R_1, \ldots, I_m, R_m$ that we constructed is in $\mathcal{T}_k/\equiv$, and the map f is well-defined.

We now need to define its inverse $g : \mathcal{T}_k/\equiv \to \mathbf{Car}(\boxplus^n)_{\widehat{\mathbf{0}}}^{\widehat{\mathbf{k}}}$. Take a sequence of sets $I_1, R_1, \ldots, I_m, R_m$ that is alternating and has exactly k responses from each process. We construct a sequence of cubes $c_0, \ldots, c_{2m}$ by induction as follows. We start with $c_0 = \widehat{\mathbf{0}}$. Assume that $c_{2\ell}$ has been defined. Then:

– $c_{2\ell+1}$ is the cube obtained from $c_{2\ell}$ by removing the hats from each component $j \in I_{\ell+1}$. The natural number inside the component remains unchanged.

– $c_{2\ell+2}$ is the cube obtained from $c_{2\ell+1}$ by incrementing the value of each component $j \in R_{\ell+1}$, and adding a hat to them.

We now prove that this is indeed a carrier sequence. Checking that $c_{2\ell} \lhd^- c_{2\ell+1}$ is straightforward: just use the face maps ∂_i^- with the indexes corresponding to the components where hats must be added. Checking that $c_{2\ell+1} \rhd^+ c_{2\ell+2}$ is more tricky: we must make sure that we never increment a component that already has a hat, since face maps cannot do that. To prove this, we use the fact that the sequence $I_1, R_1, \ldots, I_m, R_m$ is alternating, and in particular that no two responses can occur consecutively. For each $j \in R_{2\ell+2}$ we look at the largest $r < 2\ell + 1$ such that $j \in I_r$ (it must exist because of (i)). By (iii), there is no s such that $r \leq s < 2\ell + 2$ and $j \in R_s$. Then the invocation I_r has removed the hat from the j-th component, and this component has not been changed since then. Therefore each $j \in R_{2\ell+2}$ is the index of a component without hat of $c_{2\ell+1}$, and we can choose the corresponding face maps ∂_i^+ so that $c_{2\ell+1} \rhd^+ c_{2\ell+2}$. Finally, we also need to check that $c_{2m} = \widehat{\mathbf{k}}$, which is the case because each component will be incremented k times, and because the trace is complete.

To show that $f \circ g = \mathsf{id}$, take a sequence $s = I_1, R_1, \dots, I_m, R_m$, and its image $g(s) = c_1, \dots, c_{2m-1}$. We can prove by an easy induction that the set of indexes without hats of $c_{2\ell+1}$ is given by the formula $((((I_1 \setminus R_1) \cup I_2) \setminus R_2) \cup \dots) \cup I_{\ell+1}$, and the one of $c_{2\ell+2}$ is $(((((I_1 \setminus R_1) \cup I_2) \setminus R_2) \cup \dots) \cup I_{\ell+1}) \setminus R_{\ell+1}$. In other words, the sets S_r that we use in the construction of f satisfy the equations $S_{2\ell+1} = S_{2\ell} \cup I_{\ell+1}$ and $S_{2\ell+2} = S_{2\ell+1} \setminus R_{\ell+1}$. Then $f \circ g(s)$ will give the sets $S_{2\ell-1} \setminus S_{2\ell-2} = I_\ell$ and $S_{2\ell-1} \setminus S_{2\ell} = R_\ell$.

Conversely, let $\overline{\mathbf{c}} = c_1, \dots, c_{2m-1}$ be a carrier sequence, and let $f(\overline{\mathbf{c}}) = I_1, R_1, \dots, I_m, R_m$. By definition, we have $I_\ell = S_{2\ell-1} \setminus S_{2\ell-2}$ and $R_\ell = S_{2\ell-1} \setminus S_{2\ell}$, with $S_0 = S_{2m} = \varnothing$. Let $g \circ f(\overline{\mathbf{c}}) = c'_1, \dots, c'_{2m-1}$ be the sequence of cubes obtained by the definition of g. We pose $c_0 = c'_0 = \widehat{\mathbf{0}}$, and we prove by induction on r that for all r, $c_r = c'_r$. This is true for $r = 0$. Assume that $c'_{2\ell} = c_{2\ell}$. Then:

- $c'_{2\ell+1}$ is obtained from $c'_{2\ell} = c_{2\ell}$ by removing the hats from each component $j \in I_{\ell+1}$. Moreover, since $c_{2\ell} \lhd^- c_{2\ell+1}$, the cube $c_{2\ell+1}$ is also obtained from $c_{2\ell}$ by removing hats. The set of indexes where hats have been removed is precisely $S_{2\ell+1} \setminus S_{2\ell}$, which is equal to $I_{\ell+1}$ by definition.
- $c'_{2\ell+2}$ is obtained from $c'_{2\ell+1} = c_{2\ell+1}$ by incrementing each component $j \in R_{\ell+1}$ and adding hats. Since $c_{2\ell+1} \rhd^+ c_{2\ell+2}$, the cube $c_{2\ell+2}$ is also obtained from $c_{2\ell+1}$ by incrementing some indexes and adding hats. The set of indexes where hats have been added is $S_{2\ell+1} \setminus S_{2\ell+2} = R_{\ell+1}$. □

4.2.2 Chromatic subdivisions via partial cube chains

The bijections of the previous section explain the computational intuition behind the various notions of paths, but they do not really reflect the topological structure of the space of paths on an HDA. Indeed, as it is shown in Ziemiański's paper about cube chains [119], one can link cube chains together in order to capture the continuous deformation of paths that occurs on HDAs. Similarly, if we want to construct protocol complexes from HDA semantics, we need to consider not only the sets of executions (i.e., the set of facets of the protocol complex), but also how those facets are glued together (i.e., the local *views* of the processes). This is the same issue as for the test-and-set and compare-and-swap examples presented in the introduction of this chapter.

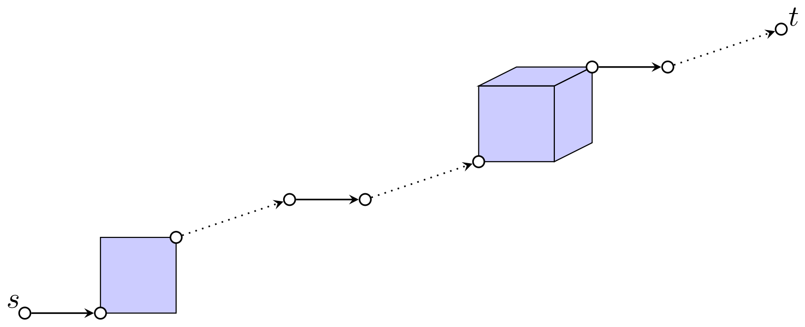
In this section, we focus on the equivalence between cube chains and set-sequential traces. The typical example of a set-linearizable object is the immediate-snapshot object. Its particularity is that each set-sequential execution trace (up to equivalence) produces a distinct global state. Thus, we have a bijection between the set of set-sequential traces, and the set of facets of the immediate-snapshot protocol complex. By Theorem 4.15, this means that we have a bijection between the facets of the standard chromatic subdivision, and the set of cube chains. Our goal in this section is to recover the full structure of the standard chromatic subdivision, that is, explain how these facets are glued together. To do so, we introduce the new notion of *partial cube chains*.

Definition 4.18. Let K be a pre-cubical set and $s, t \in K_0$ two of its vertices. A *partial cube chain* from s to t in K is a sequence of cubes $\overline{\mathbf{c}} = (c_1, \dots, c_\ell)$ where $c_i \in K_{n_i}$ $(n_i > 0)$ such that:

- There is a (possibly empty) edge path from s to $\partial^-(c_1)$.
- There is a (possibly empty) edge path from $\partial^+(c_\ell)$ to t.
- For $1 \leq i \leq \ell - 1$, there is a (possibly empty) edge path from $\partial^+(c_i)$ to $\partial^-(c_{i+1})$.

We write $\mathbf{PCh}(K)_s^t$ for the set of partial cube chains from s to t.

Note that in particular, cube chains are partial cube chains, i.e., $\mathbf{Ch}(K)_s^t \subseteq \mathbf{PCh}(K)_s^t$. The length and dimension of partial cube chains are defined as for total ones. A partial cube chain can be pictured as follows, where dotted lines represent the existence of a path.



Definition 4.19. Given two partial cube chains $\overline{\mathbf{c}}, \overline{\mathbf{c}}' \in \mathbf{PCh}(K)_s^t$, we say that $\overline{\mathbf{c}}'$ *extends* $\overline{\mathbf{c}}$, written $\overline{\mathbf{c}} \preccurlyeq \overline{\mathbf{c}}'$, whenever:

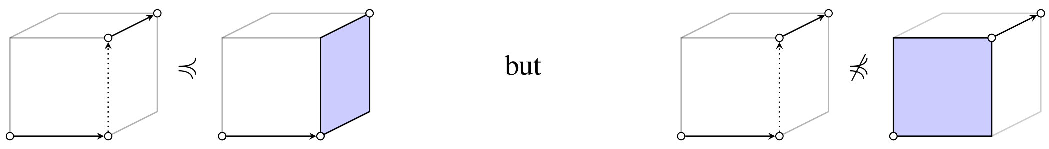
– Either $\overline{\mathbf{c}}' = (c_1, \dots, c_\ell)$ and $\overline{\mathbf{c}} = (c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_\ell)$, i.e., $\overline{\mathbf{c}}$ is obtained from $\overline{\mathbf{c}}'$ by removing one cube.

– Or $\overline{\mathbf{c}}' = (c_1, \dots, c_{i-1}, c_i', c_{i+1} \dots, c_\ell)$ and $\overline{\mathbf{c}} = (c_1, \dots, c_{i-1}, c_i, c_{i+1}, \dots, c_\ell)$, where $c_i = \partial_k^+(c_i')$ for some $k < n_i'$.

By taking the reflexive and transitive closure of this relation (still written $\preccurlyeq$), we obtain a partial order on $\mathbf{PCh}(K)_s^t$ (antisymmetry is proved by looking at the length of the chains).

Remark 4.20. Note that in the second case of Definition 4.19, c_i is only allowed to be a forward face of c_i'. This introduces some asymmetry in the relation, as illustrated in Example 4.21. This point will be crucial when we want to connect this to the simplicial chromatic subdivision that arises in immediate snapshot protocols. Indeed, a process P views a process Q whenever Q runs either before or concurrently, and doesn't view it when Q runs after P.

Example 4.21. In the 3-dimensional cube $\square^3$:



We will now show that the partial cube chains in $\square^n$, equipped with the partial order $\preccurlyeq$, are order isomorphic to the poset of simplices of the standard chromatic subdivision. First, we recall the definition of the one-round standard chromatic subdivision (cf. Section 1.2.3).

Definition 4.22. The *combinatorial n-simplex* Δ^n is the abstract simplicial complex with $[n]$ as the set of vertices and all subsets of $[n]$ as simplices. Thus, Δ^n is of dimension n.

Definition 4.23. The standard chromatic subdivision of Δ^n, written $\mathsf{ChSub}(\Delta^n)$, is the following abstract simplicial complex:

– Its set of vertices is $\{(i, X_i) \mid X_i \in [n], i \in X_i\}$.

– The set $\{(i_0, X_{i_0}), \dots, (i_k, X_{i_k})\}$ is a k-simplex iff

(1) It can be indexed so that $X_{i_0} \subseteq \dots \subseteq X_{i_k}$, and

(2) If $i_\ell \in X_{i_m}$ then $X_{i_\ell} \subseteq X_{i_m}$.

Recall the definition of the standard $(n+1)$-cube $\square^{n+1}$ (see Definition 4.4). We write $\mathbf{s} := - \cdots -$ for its source, and $\mathbf{t} := + \cdots +$ for its target. We can now state the main theorem of this section:

Theorem 4.24. *The poset of partial cube chains in the combinatorial $(n+1)$-cube is order isomorphic to the face poset of the n-dimensional standard chromatic subdivision:*

$$(\mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}}, \preccurlyeq) \;\simeq\; (\mathsf{ChSub}(\Delta^n), \subseteq)$$

Before we prove it, let us show a helper lemma about what partial cube chains in the cube look like:

Lemma 4.25. *In a partial cube chain $\overline{\mathbf{c}} = (c_1, \dots, c_\ell) \in \mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}}$, whenever a cube c_k has a '0' or '+' symbol at a position i, all the subsequent cubes also have a '+' symbol at position i.*

Proof. We prove the lemma for total cube chains. Since a partial cube chain can always be extended to a total one (by filling the holes with the paths that are required to exist by definition), the result follows for all partial cube chains. The endpoint $v = \partial^+(c_k)$ must have a '+' symbol at position i. Moreover, we must also have $v = \partial^-(c_{k+1})$ since we are dealing with a cube chain. As a result, c_{k+1} can only have a '+' symbol at position i (otherwise $\partial^-(c_{k+1})$ would have a '$-$' at position i). By iterating this reasoning, all the subsequent cubes must also have a '+'. □

Corollary 4.26. *Let $\overline{\mathbf{c}} = (c_1, \dots, c_\ell) \in \mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}}$. For every $i \in [n]$, there is at most one cube c_k that has symbol 0 at position i.*

Proof. Let c_k be the first cube (if any) that has a '0' symbol at position i. By Lemma 4.25, all the subsequent cubes must have a '+'. □

Remark 4.27. The computational intuition behind this is that $\square^{n+1}$ is interpreted as the trace space for $n+1$ processes, each of which performs one set-linearizable operation (such as a call to the immediate-snapshot object). A cube chain in that space represents one possible execution. Running the process i corresponds to advancing in the i-th direction along the cube. Corollary 4.26 states that in such an execution, each process runs only once.

If c is a cube in $\square^{n+1}$, we write $\mathsf{proc}_c \subseteq [n]$ for the set of indexes where c has symbol '0', and $\mathsf{view}_c \subseteq [n]$ for the set of indexes where c has symbol '0' or '+'.

Lemma 4.28. *Let $\overline{\mathbf{c}} = (c_1, \dots, c_\ell)$ be a partial cube chain in $\square^{n+1}$. If $k \leq k'$, then $\mathsf{view}_{c_k} \subseteq \mathsf{view}_{c_{k'}}$.*

Proof. This is a direct consequence of Lemma 4.25 □

Definition 4.29. We define a map $f : \mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}} \to \mathsf{ChSub}(\Delta^n)$ as follows. Let $\overline{\mathbf{c}} = (c_1, \dots, c_\ell)$ be a partial cube chain. Then $f(\overline{\mathbf{c}})$ is the simplex

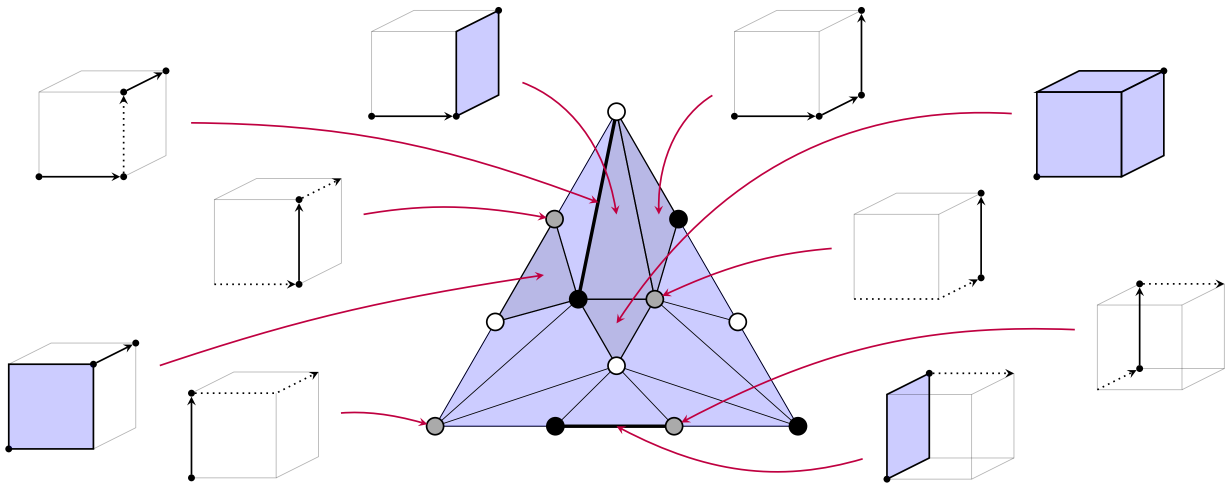
$$f(\overline{\mathbf{c}}) = \bigcup_{1 \leq k \leq \ell} \{(i, \mathsf{view}_{c_k}) \mid i \in \mathsf{proc}_{c_k}\}$$

Let us show that it is indeed a simplex of $\mathsf{ChSub}(\Delta^n)$. First, $i \in \mathsf{view}_{c_k}$ since $i \in \mathsf{proc}_{c_k} \subseteq \mathsf{view}_{c_k}$. The views in $f(\overline{\mathbf{c}})$ can be totally ordered by inclusion thanks to Lemma 4.28. Finally, suppose (i, view_{c_k}) and (j, view_{c_m}) are two elements of $f(\overline{\mathbf{c}})$ such that $i \in \mathsf{view}_{c_m}$. Then c_m has a '0' or '+' symbol at position i, and by Lemma 4.25, all the subsequent cubes must have a '+' at position i. Since $i \in \mathsf{proc}_{c_k}$, c_k cannot be after c_m, i.e., $k \leq m$. By Lemma 4.28, $\mathsf{view}_{c_k} \subseteq \mathsf{view}_{c_m}$.

Lemma 4.30. *The dimension of $f(\overline{\mathbf{c}})$ is* $\dim(f(\overline{\mathbf{c}})) = \mathsf{length}(\overline{\mathbf{c}}) - 1$.

Proof. Let $\overline{\mathbf{c}} = (c_1, \dots, c_\ell)$ be of type $(n_1, \dots, n_\ell)$. The sets proc_{c_i} are disjoint by Corollary 4.26. Thus, $|f(\overline{\mathbf{c}})| = \sum_i |\mathsf{proc}_{c_i}| = \sum_i n_i = \mathsf{length}(\overline{\mathbf{c}})$ □

The picture below illustrates the map f for $n = 2$: partial cube chains in the 3-dimensional cube correspond to the simplexes of the 2-dimensional chromatic subdivision. In particular, total cube chains, which are of length 3, correspond to the facets of $\mathsf{ChSub}(\Delta^2)$, of dimension 2.



Definition 4.31. We now define $g : \mathsf{ChSub}(\Delta^n) \to \mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}}$. Let $X = \{(i_0, X_{i_0}), \dots, (i_m, X_{i_m})\}$ be a simplex in $\mathsf{ChSub}(\Delta^n)$, indexed such that $X_{i_0} \subseteq \ldots \subseteq X_{i_m}$. We split this sequence at the indexes where the inclusion is strict. Let $(I_k)_{1 \leq k \leq \ell}$ be the (unique) partition of $\{i_0, \dots, i_m\}$ such that:

– For $j, j' \in I_k$, $X_j = X_{j'}$. We write this common view X_{I_k}.

– The inclusion $X_{I_k} \subsetneq X_{I_{k+1}}$ is strict.

We can now construct a partial cube chain $g(X) = (c_1, \dots, c_\ell)$, where c_k has symbol '0' at all positions $j \in I_k$; symbol '$+$' at all positions $j \in (X_{I_k} \setminus I_k)$; and symbol '$-$' at the remaining positions.

To check that this is indeed a partial cube chain, we need to show that there is a (possibly empty) path from $\partial^+(c_k)$ to $\partial^-(c_{k+1})$. The only effect a path in the cube can have on vertices is to turn '$-$' symbols into '$+$' symbols, so we need to check that whenever $\partial^+(c_k)$ has a '$+$' symbol at some position, then $\partial^-(c_{k+1})$ also does. $\partial^+(c_k)$ has a '$+$' symbol exactly at the positions $j \in X_{I_k}$. Since $X_{I_k} \subseteq X_{I_{k+1}}$, c_{k+1} has either a '0' or a '$+$' at those positions, so we need to show that it cannot be a '0', i.e., $I_{k+1} \cap X_{I_k} = \varnothing$. Suppose there is $j \in I_{k+1}$ such that $j \in X_{I_k}$, then by condition (2) of Definition 4.23, we would have $X_{I_{k+1}} \subseteq X_{I_k}$ which is impossible since the inclusion is strict.

Lemma 4.32. *f and g are inverse of each other.*

Proof. It is straightforward to check that $f \circ g(X) = X$. We take the notations from Definition 4.31, with $g(X) = (c_1, \dots, c_\ell)$. Then $\mathsf{proc}_{c_k} = I_k$ and $\mathsf{view}_{c_k} = X_{I_k}$, and we get

$$f \circ g(X) = \bigcup_{1 \leq k \leq \ell} \{(i, X_{I_k}) \mid i \in I_k\}$$

Since the sets $(I_k)_{1 \leq k \leq \ell}$ form a partition of $\{i_0, \dots, i_m\}$, and $X_{I_k} = X_i$ for $i \in I_k$, we get $f \circ g(X) = X$.

Now we show that $g \circ f(\overline{\mathbf{c}}) = \overline{\mathbf{c}}$. By Lemma 4.28, the views that appear in $f(\overline{\mathbf{c}})$ are ordered according to the ordering of the cubes of $\overline{\mathbf{c}}$: $\mathsf{view}_{c_1} \subsetneq \ldots \subsetneq \mathsf{view}_{c_\ell}$. Moreover, all those inclusions are strict: since cube chains do not contain 0-cubes, c_{k+1} must have at least one position with symbol '0', and that position cannot be in view_{c_k} (otherwise it would be a '$+$'). Thus, the corresponding partition (I_k) is $I_k = \mathsf{proc}_{c_k}$ with its associated view $X_{I_k} = \mathsf{view}_{c_k}$, and applying g gives back the original chain $\overline{\mathbf{c}}$. □

Lemma 4.33. *For all* $\overline{\mathbf{c}}, \overline{\mathbf{c}}' \in \mathbf{PCh}(\square^{n+1})^{\mathbf{t}}_{\mathbf{s}}$, $\overline{\mathbf{c}} \preccurlyeq \overline{\mathbf{c}}'$ *iff* $f(\overline{\mathbf{c}}) \subseteq f(\overline{\mathbf{c}}')$.

Proof. Assume $\overline{\mathbf{c}} \preccurlyeq \overline{\mathbf{c}}'$. We treat both cases of Definition 4.19.

– $\overline{\mathbf{c}}'$ is obtained by adding one cube to $\overline{\mathbf{c}}$: since view_{c_k} and proc_{c_k} only depend on the cube and not on the rest of the chain, we trivially get $f(\overline{\mathbf{c}}) \subseteq f(\overline{\mathbf{c}}')$.

– Or $\overline{\mathbf{c}}$ and $\overline{\mathbf{c}}'$ only differ by one cube, say c_k and c'_k, such that $c_k = \partial^+_i(c'_k)$, i.e., a '0' in c'_k has been replaced by a '$+$' in c_k. Thus, $\mathsf{view}_{c_k} = \mathsf{view}_{c'_k}$ and $\mathsf{proc}_{c_k} \subseteq \mathsf{proc}_{c'_k}$, from which we deduce $f(\overline{\mathbf{c}}) \subseteq f(\overline{\mathbf{c}}')$.

Conversely, we show that if $X \subseteq X'$ then $g(X) \preccurlyeq g(X')$ (and conclude by Lemma 4.32). We treat the case when $|X'| = |X| + 1$, the general result follows by iterating the same process. Assume $X = \{(i_0, X_{i_0}), \ldots, (i_m, X_{i_m})\}$ and X' has one more element (j, X_j). Let $(I_k)_{1 \leq k \leq \ell}$ be the partition that arises from X in Definition 4.31, giving a cube chain $g(X) = (c_1, \ldots, c_\ell)$. We distinguish two cases:

– Either X_j is equal to one of the X_{I_k}, say for $k = k_0$. Then, when computing $g(X')$, we will get a partition $(I'_k)_{1 \leq k \leq \ell}$ such that $I'_{k_0} = I_{k_0} \cup \{j\}$ and $I'_k = I_k$ for $k \neq k_0$. This will give a cube chain $g(X') = (c_1, \ldots, c'_{k_0}, \ldots, c_\ell)$ that differs from $g(X)$ only by one cube c'_{k_0}. Moreover by definition of g, c'_{k_0} will have a '0' symbol at position j, whereas c_{k_0} has a '$+$' since $j \in X_j = X_{I_{k_0}}$. All other positions are the same, thus c_{k_0} is a forward face of c'_{k_0}, i.e., $g(X) \preccurlyeq g(X')$.

– Otherwise, X_j is different to all of the X_{I_k}. Then the partition (I'_k) will be the same as (I_k) with one more set $\{j\}$ added to it. Thus, $g(X')$ will have the same cubes as $g(X)$, plus one additional 1-cube. That is, $g(X) \preccurlyeq g(X')$. □

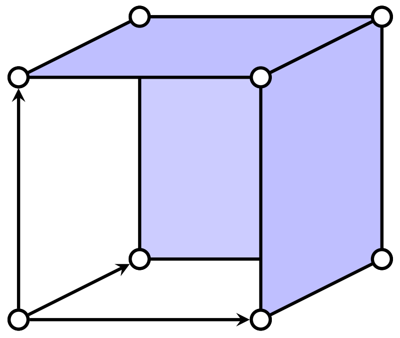
This concludes the proof of Theorem 4.24. Thus, we were able to recover the notion of *view* from distributed computability, using only notions from directed topology. In classic distributed computability, there are usually two intuitions about the protocol complex: the facets (of dimension n) correspond to global executions, and the vertices (if dimension 0) correspond to the local view of one process. The simplexes of other dimensions are usually not attributed a clear meaning. Of course, the computational meaning of those "partial executions" is not yet well-understood, but this correspondence might give some new insights about the protocol complex.

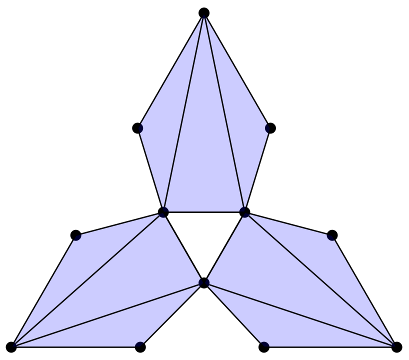
4.3 Future work and open questions

The work presented in this chapter is still very incomplete. Theorem 4.24 on partial cube chains can be extended in many ways. More importantly, the computational meaning of this construction is not completely understood yet: cube chains describe set-sequential executions in general, without making assumptions about whether we are running an immediate-snapshot object or any other set-linearizable object. So, in what sense is the notion of *partial* cube chains related to immediate-snapshot? Can we find some different ways to link the cube chains together in order to produce other protocol complexes?

Partial cube chains on other HDAs

One interesting thing that we can do, is take the construction of Section 4.2.2, and apply it to other HDAs rather than $\square^n$. For instance, the HDA modeling the test-and-set object is depicted below (left). It is the combinatorial version of the directed space for test-and-set that we mentioned in the introduction.





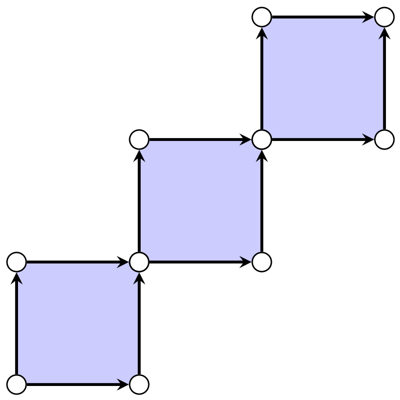
If we look at the set of partial cube chains on this HDA, equipped with the extension partial order $\preccurlyeq$, we obtain the simplicial complex depicted on the right. Surprisingly, it is quite similar topologically to the protocol complex for test-and-set, in the sense that it is homotopy-equivalent to a circle. However, it has more triangles than the one depicted in the introduction. It is also embedded on a chromatic subdivision of a simplex, which might suggest that it could be a protocol complex obtained by combining test-and-set and immediate-snapshot. Moreover, as in the case of directed spaces, the compare-and-swap object also corresponds to the same HDA, in which case this construction does not give the expected protocol complex at all.

It seems that the simplicial complex that our construction produces is a kind of "intermediate step" towards the protocol complex. It describes the information acquired by the processes thanks to the scheduling that occurred, but it does not say anything about the input and output values that a process has seen during the execution. So, in order to get the real protocol complex, one would need to add this additional information; for test-and-set it would collapse the equivalent facets, and for compare-and-swap it would separate the triangles.

This kind of picture, where protocol complexes are embedded on a chromatic subdivision with some missing facets, appear in two different settings in the literature to our knowledge. One of them is the *refined tasks* of Castañeda, Rajsbaum and Raynal [18], that we mentioned briefly in Chapter 2. The idea of refined tasks is to include scheduling information in addition to the output/output relation of classical tasks. They were introduced in order to provide a task specification mechanism that is as expressive as interval-linearizability. The other one is called *affine tasks* [82], which are defined as sub-complexes of the second iteration of the chromatic subdivision. They are used to provide an Asynchronous Computability Theorem for various adversarial models in a read/write memory setting.

Iterated chromatic subdivisions

An important feature of the immediate-snapshot model is that we can iterate it in order to subdivide the input complex further. It seems that our current definition of partial cube chains may not be suitable to produce iterated chromatic subdivisions. A first idea would be to look at the poset of partial cube chains from $\widehat{0}$ to $\widehat{k}$ in the stack of cubes $\boxplus^n$, as we did in Section 4.2.1. However, this does not model the round-based structure of the iterated immediate snapshot model, since in $\boxplus^n$ a process can start the second “round” before the other processes are finished with the first one. A better option is to look at partial cube chains in the following HDA (for 2 processes and 3 rounds):



But even in this case, we do not get the right simplicial complex. The reason is that partial cube chains does not keep track of process identity. For example, in the picture above, the three horizontal actions are performed by one process, and the three vertical actions are performed by the other process. In order to obtain the iterated immediate-snapshot protocol complex, one would have to take this into account in the definition of partial cube chains, by requiring that whenever some cube in the chain is missing, then the corresponding processes never appear again in the future.

Non-immediate write-snapshot object

The (non-immediate) write-snapshot object is also very well-studied in the distributed computing literature. Its protocol complex includes the chromatic subdivision, as well as some extra simplexes that do not alter the topology of the input complex. It was shown in [18] that the right way to specify this object is using interval-linearizability, since its specification cannot be expressed using set-linearizability. Thanks to Theorem 4.17, it is easy to see that we have a bijection between the set of carrier sequences in $\square^{n+1}$, and the facets of the n-dimensional protocol complex for non-immediate write-snapshot.

As in Section 4.2.2, the goal would now be to find a way to interpret the lower-dimensional simplices of the protocol complex, as well as the face relations between them. However this seems quite tedious to do, and we currently do not have a candidate for a suitable notion of “partial carrier sequence”.

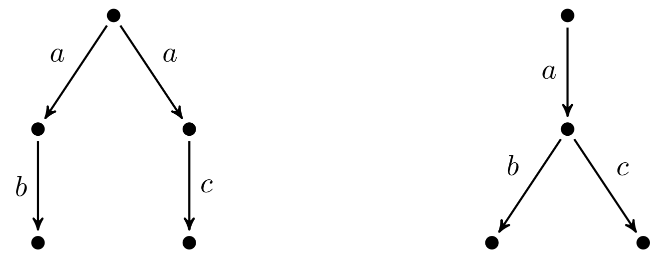
Modeling the space of directed paths

Originally, cube chains were introduced by Ziemiański in [119] as a model for the space of directed paths in the geometric realization of a pre-cubical set. The important result in this paper is that the combinatorial model that he builds using cube chains (which is a CW-complex instead of a simplicial complex) is homotopy equivalent to the (continuous) path space in the corresponding d-space.

In our case, we can already see that this property does not hold in our partial cube chains model, by looking at the test-and-set example on page 172. Indeed, the trace space of the HDA on the left has three connected components, since there are three non-homotopic paths. However, the simplicial complex on the right has the homotopy type of a circle, and only one connected component! After discussions with Krzysztof Ziemiański, it seemed that a possible way to fix this issue would be to change the definition of partial cube chains (Definition 4.18) and replace each occurrence of “there exists a path from x to y” by “the trace-space between x and y is contractible”. On the standard n-cube $\square^n$, the two definitions are equivalent since whenever there exists a path between two points, the trace space is always contractible. If we apply this definition to the test-and-set example, this would remove the three middle vertices of the simplicial complex, which would create three connected components. However, what we get is not a simplicial complex anymore, since we remove vertices but not the triangles and edges that contain them.

HDA semantics for non-linearizable objects

The trace-based semantics that we used in Chapter 2 specifies concurrent objects by giving the set of all execution traces that they can produced. In the setting of labeled transition systems, it is well known that trace equivalence is not always the good notion to compare program behaviors. For instance, in the context of non-determinism, two systems might have distinct observable behaviors but still produce the same sets of traces. To distinguish them, one must rely on notions such as *bisimilarity*.



HDAs can be viewed as labeled transition systems with higher-dimensional cells, which encode whether or not some operations may be performed concurrently. Thus, using HDAs, we hope to be able to exhibit that kind of distinction between concurrent programs, that trace semantics cannot express. The idea would be to replace Definition 2.22 by a definition based on an HDA. The three bijections of Section 4.2.1 suggest that we could capture the notions of sequential, set-sequential and interval specifications by labeling the sets of edge paths, cube chains and carrier sequences (respectively) on an HDA with output values. As a tentative definition, suppose given a set $\mathcal{A}$ of *actions*, containing a special value $\bot \in \mathcal{A}$. In practice, an action could contain information such as the object and method being called, as well as the input value of the call. We write $\mathbf{Ch}(K)_s$ for the set of cube chains with source s and any target.

Definition 4.34. An *HDA* set-sequential specification is given by a tuple $(K, s, \lambda, \mathcal{V}, \Delta)$ where:

- K is a pre-cubical set,
- $s \in K_0$ is the *source vertex*,
- λ is a map from K to sequences of actions. To a n-dimensional cube $x \in K_n$, it associates a sequence $\lambda(x) = (\lambda_0(x), \dots, \lambda_i(x), \dots)$ such that $|\{i \in \mathbb{N} \mid \lambda_i(x) \neq \bot\}| = n$. Moreover, for all cube x and direction j, we require $\lambda(\partial_j^+(x)) = \lambda(\partial_j^-(x))$.
- $\mathcal{V}$ is a set of *output values*, containing the special value $\bot$,
- Δ associates to each cube chain $c_1, \dots, c_\ell \in \mathbf{Ch}(K)_s$ a set of ℓ-tuples in $(\mathcal{V}^{\mathbb{N}})^\ell$. The r-th component of such a tuple (written $\mathsf{resp}(c_r)$) is a sequence of values assigning responses to the participating processes in c_r. We require $\{j \in \mathbb{N} \mid \mathsf{resp}(c_r)_j \neq \bot\} = \{j \in \mathbb{N} \mid \lambda_j(c_r) \neq \bot\}$.

An interval-specification would be a similar definition, replacing the set of cube chains $\mathbf{Ch}(K)_s$ by carrier sequences $\mathbf{Car}(K)_s$ (then capturing the sets of participating processes becomes a bit cumbersome). To get a general notion of HDA specification (analogous to our definition of concurrent specifications in Chapter 2), we would add the expansion property, which amounts to requiring that Δ be monotonic with respect to inclusion of carrier sequences. Ultimately, the goal would be to have the following correspondences:

- HDA specifications on $\boxplus^n$ correspond to concurrent specifications (Definition 2.22),
- HDA specifications on $\square^n$ correspond to one-shot concurrent specifications (Definition 2.55).

On other HDAs, such a definition would provide us with a handy way of dealing with object that include some level of synchronization: locks, barriers, and so on. Lastly, we could also consider HDAs of infinite dimension, which would allow us to deal with an unbounded number of processes.

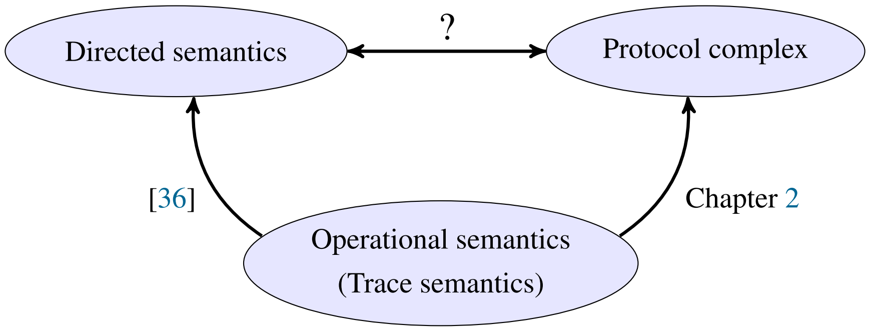
As we mentioned above, the main motivation for moving from trace semantics to HDA semantics is that HDAs can be compared using various notions of *bisimulation* [33, 32]. This richer structure should allow for a better understanding of what concurrent programs are computing. For instance, it would be interesting to see whether the Galois connection of Theorem 2.32 turns into an adjunction between some categories of HDA specifications.

Conclusion

The starting point of this thesis was the paper by Goubault, Mimram and Tasson [55], which shows that we can recover the immediate-snapshot protocol complex by looking at di-homotopy classes of paths in the associated directed space. Our initial goal was to extend this correspondence between directed topology and the protocol complex approach to other classes of concurrent objects.

By looking at other examples such as test-and-set, it became clear that there was always a one-to-one correspondence between the di-homotopy classes of paths in the directed semantics and the facets of the protocol complex. But we could not figure out how to recover the "gluing" between these facets. One of the first results that we came up with was the construction of the chromatic subdivision via partial cube chains (Theorem 4.24). However, we could not understand the meaning behind this construction: the notion of cube chain alone does not have any relationship with the immediate-snapshot object. Indeed, the different kinds of paths on HDAs are related to the schedulings that can occur, but they do not give information about the values exchanged between processes.

We decided that, instead of trying to link directly the two geometric approaches together, we should first understand how they can both be derived from a more concrete operational semantics. In the directed space semantics as presented in [36], it is clear how to obtain a directed space from any program, by providing the semantics of the basic objects that we use. On the other hand, the protocol complex in [64] is defined only for some particular objects, and assuming a round-based structure.



So we began the work that we presented in Chapter 2, which proved to be much more challenging that we initially expected. Indeed, after discussing with Sergio Rajsbaum about his work on interval-linearizability [18], we realized that the objects of interest for asynchronous computability are usually of a more complex nature than the linearizable ones that appear in practical implementations. As it turned out, this was a very fruitful endeavor, since we discovered that the different variants of linearizability actually correspond to the various notions of paths on HDAs (Section 4.2.1). Thus, we managed to understand the link between cube chains and the immediate-snapshot object: they both describe set-sequential executions.

The work presented in Chapter 3 was also a result of our collaboration with Sergio Rajsbaum. Studying the protocol complexes from the point of view of epistemic logic was quite illuminating, and helped us understand better what is really going on in the abstract topological proofs for impossibility. Even though the notion of knowledge has always been present in the intuitive explanations about the protocol complex construction, the relationship between the two had never been fully formalized before. Perhaps one of the key points that we have learned is that the information contained in the protocol complex is not so much the local views of the processes, but the *indistinguishability* between executions. Even though the indistinguishability relations are often derived from a notion of local view, it need not be the case in general. Hence, another way to recover the protocol complex from the directed semantics could be to equip it with some notion of indistinguishability.

In conclusion, studying the protocol complex from various different angles has given us a much better understanding of the concrete properties of programs underlying this abstract topological construction. While there are still many research directions left to explore, it seems that we now have enough tools to start investigating the relationship between the directed semantics and the protocol complexes for general non-linearizable objects. The notion of HDA specification that we sketched at the very end of Chapter 4 seems to be a quite promising line of research.

Bibliography

[1] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 334–344, 1998. doi:10.1109/LICS.1998.705669.

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. doi:10.1006/inco.2000.2930.

[3] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of choices, failures and asynchrony: The many faces of set agreement. *Algorithmica*, 62(1-2):595–629, 2012. doi:10.1007/s00453-011-9581-7.

[4] H. Attiya and S. Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002. doi:10.1137/S0097539797330689.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

[6] A. Baltag, L.S. Moss, and S. Solecki. The logic of common knowledge, public announcements, and private suspicions. In *TARK VII*, pages 43–56, 1998. doi:10.1007/978-3-319-20451-2_38.

[7] A. Baltag and B. Renne. Dynamic epistemic logic. In *The Stanford Encyclopedia of Philosophy,* see https://plato.stanford.edu/archives/win2016/entries/dynamic-epistemic/. Metaphysics Research Lab, Stanford University, 2016.

[8] Alexandru Baltag and Lawrence S. Moss. Logics for epistemic programs. *Synthese*, 139(2):165–224, 2004.

[9] F. Benavides and S. Rajsbaum. Collapsibility of read/write models using discrete morse theory. *Journal of Applied and Computational Topology*, pages 1–32, 2018. doi:10.1007/s41468-018-0011-7.

[10] O. Biran, S. Moran, and S. Zaks. A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *J. Algorithms*, 11(3):420–440, 1990. doi:10.1016/0196-6774(90)90020-F.

[11] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 2001.

[12] Thomas Bolander, Hans van Ditmarsch, Andreas Herzig, Emiliano Lorini, Pere Pardo, and Francois Schwarzentruber. Announcements to attentive agents. *Journal of Logic, Language and Information*, 25(1):1–35, 2016.

[13] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North Holland, 5 edition, 1982.

[14] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167119.

[15] Elizabeth Borowsky and Eli Gafni. Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51. ACM, 1993.

[16] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 189–198. ACM Press, 1996.

[17] A. Castañeda, Y. A. Gonczarowski, and Y. Moses. Unbeatable set consensus via topological and combinatorial reasoning. In *PODC*, pages 107–116. ACM, 2016. doi:10.1145/2933057.2933120.

[18] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

[19] A. Castañeda, Y. A. Gonczarowski, and Y. Moses. Unbeatable consensus. In *DISC*, number 8784 in LNCS, pages 91–106. Springer, 2014. doi:10.1007/978-3-662-45174-8_7.

[20] Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Generalized symmetry breaking tasks and nondeterminism in concurrent objects. *SIAM J. Comput.*, 45(2):379–414, 2016. doi:10.1137/130936828.

[21] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5):287–301, Aug 2010. doi:10.1007/s00446-010-0108-2.

[22] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Long-lived tasks. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems*, pages 439–454, Cham, 2017. Springer International Publishing.

[23] Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. The concurrent game semantics of probabilistic PCF. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 215–224, 2018. doi:10.1145/3209108.3209187.

[24] 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 ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[25] C. Dégremont, B. Löwe, and A. Witzel. The synchronicity of dynamic epistemic logic. In *TARK XIII*, pages 145–152. ACM, 2011. doi:10.1145/2000378.2000395.

[26] Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 29(9):2033–2045, 2018. doi:10.1109/TPDS.2018.2809551.

[27] H. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*. Springer, 2007. doi:10.1007/978-1-4020-5839-4.

[28] Jérémy Dubut. *Directed homotopy and homology theories for geometric models of true concurrency. (Théories homotopiques et homologiques dirigées pour des modèles géométriques de la vraie concurrence)*. PhD thesis, University of Paris-Saclay, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01590515.

[29] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a byzantine environment: Crash failures. *Inf. Comput.*, 88(2):156–186, 1990. doi:10.1016/0890-5401(90)90014-9.

[30] Samuel Eilenberg. Ordered topological spaces. *American Journal of Mathematics*, 63(1):39–45, 1941. URL: http://www.jstor.org/stable/2371274.

[31] Uli Fahrenberg, Christian Johansen, Georg Struth, and Ratan Badahur Thapa. Pomset languages of Higher-Dimensional Automata (unpublished).

[32] Uli Fahrenberg and Axel Legay. History-preserving bisimilarity for higher-dimensional automata via open maps. *Electr. Notes Theor. Comput. Sci.*, 298:165–178, 2013. doi:10.1016/j.entcs.2013.09.012.

[33] Ulrich Fahrenberg. A category of higher-dimensional automata. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 187–201, 2005. doi:10.1007/978-3-540-31982-5_12.

[34] Lisbeth Fajstrup. Dipaths and dihomotopies in a cubical complex. *Adv. Appl. Math.*, 35(2):188–206, 2005. doi:10.1016/j.aam.2005.02.003.

[35] Lisbeth Fajstrup, Éric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. Trace spaces: An efficient new technique for state-space reduction. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 274–294, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[36] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. *Directed Algebraic Topology and Concurrency*. Springer, 2016. doi:10.1007/978-3-319-15398-8.

[37] Lisbeth Fajstrup, Eric Goubault, and Martin Raußen. Detecting deadlocks in concurrent systems. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, pages 332–347, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[38] Lisbeth Fajstrup, Martin Raußen, and Eric Goubault. Algebraic topology and concurrency. *Theor. Comput. Sci.*, 357(1-3):241–278, 2006. doi:10.1016/j.tcs.2006.03.022.

[39] Lisbeth Fajstrup, Martin Raußen, Eric Goubault, and Emmanuel Haucourt. Components of the fundamental category. *Applied Categorical Structures*, 12(1):81–108, 2004. doi:10.1023/B:APCS.0000013812.75342.de.

[40] Ky Fan. Simplicial maps from an orientable n-pseudomanifold into S^m with the octahedral triangulation. *Journal of Combinatorial Theory*, 2(4):588–602, 1967. doi:https://doi.org/10.1016/S0021-9800(67)80063-2.

[41] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379 – 4398, 2010. European Symposium on Programming 2009.

[42] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

[43] Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

[44] Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In Shlomi Dolev, editor, *Distributed Computing*, pages 329–338. Springer Berlin Heidelberg, 2006.

[45] Seth Gilbert and Wojciech Golab. Making sense of relativistic distributed systems. In Fabian Kuhn, editor, *Distributed Computing, DISC 2015*, volume 8784 of *LNCS*, pages 361–375. Springer Berlin Heidelberg, 2014.

[46] E. Goubault and S. Rajsbaum. A simplicial complex model of dynamic epistemic logic for fault-tolerant distributed computing. Technical report, arXiv:1703.11005, 2017.

[47] E. Goubault and S. Rajsbaum. Models of fault-tolerant distributed computation via dynamic epistemic logic. Technical report, arXiv:1704.07883, 2017.

[48] Eric Goubault and Emmanuel Haucourt. Components of the fundamental category II. *Applied Categorical Structures*, 15(4):387–414, 2007. doi:10.1007/s10485-007-9082-7.

[49] Eric Goubault and Thomas P. Jensen. Homology of higher dimensional automata. In *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, pages 254–268, 1992. doi:10.1007/BFb0084796.

[50] Éric Goubault, Marijana Lazić, Jérémy Ledent, and Sergio Rajsbaum. Wait-free solvability of equality negation tasks. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary.*, pages 21:1–21:16, 2019. doi:10.4230/LIPIcs.DISC.2019.21.

[51] Éric Goubault, Marijana Lazić, Jérémy Ledent, and Sergio Rajsbaum. A dynamic epistemic logic analysis of the equality negation task. *Dynamic Logic: New Trends and Applications, DaLi 2019*, to appear.

[52] Éric Goubault, Jérémy Ledent, and Samuel Mimram. Concurrent specifications beyond linearizability. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018*, pages 28:1–28:16, 2018. doi:10.4230/LIPIcs.OPODIS.2018.28.

[53] Éric Goubault, Jérémy Ledent, and Sergio Rajsbaum. A simplicial complex model for dynamic epistemic logic to study distributed task computability. In *Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018.*, pages 73–87, 2018. doi:10.4204/EPTCS.277.6.

[54] Éric Goubault, Samuel Mimram, and Christine Tasson. Iterated chromatic subdivisions are collapsible. *Applied Categorical Structures*, 23(6):777–818, 2015. doi:10.1007/s10485-014-9383-6.

[55] Éric Goubault, Samuel Mimram, and Christine Tasson. Geometric and combinatorial views on asynchronous computability. *Distributed Computing*, 31(4):289–316, Aug 2018. doi:10.1007/s00446-018-0328-4.

[56] Marco Grandis. *Directed Algebraic Topology: Models of Non-Reversible Worlds*. New Mathematical Monographs. Cambridge University Press, 2009. doi:10.1017/CBO9780511657474.

[57] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016.

[58] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990. doi:10.1145/79147.79161.

[59] J. Havlicek. Computable obstructions to wait-free computability. *Distributed Computing*, 13(2):59–83, 2000. doi:10.1007/s004460050068.

[60] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular Verification of Concurrency-Aware Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, Proceedings*, pages 371–387, 2015.

[61] Michael Henle. *A Combinatorial Introduction to Topology*. Dover, 1983. doi:10.2307/1574757.

[62] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC*, pages 276–290. ACM, 1988. doi:10.1145/62546.62593.

[63] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[64] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.

[65] Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 324–333, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198119.

[66] Maurice Herlihy and Sergio Rajsbaum. *Algebraic topology and distributed computing: a primer*, pages 203–217. Springer Berlin Heidelberg, 1995. doi:10.1007/BFb0015245.

[67] Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing (STOC), El Paso, Texas, USA, May 4-6, 1997*, pages 589–598, 1997.

[68] Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 133–142, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277722.

[69] Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. An overview of synchronous message-passing and topology. *Electr. Notes Theor. Comput. Sci.*, 39(2):1–17, 2001. doi:10.1016/S1571-0661(05)01148-5.

[70] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999. URL: http://doi.acm.org/10.1145/331524.331529.

[71] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[72] Y. Hirai. An intuitionistic epistemic logic for sequential consistency on shared memory. In *LPAR*, pages 272–289. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-17511-4_16.

[73] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[74] Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.*, 221(1-2):393–456, 1999. doi:10.1016/S0304-3975(99)00039-0.

[75] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. doi:10.1006/inco.2000.2917.

[76] Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distrib. Comput.*, 93-94:1–9, 2016. doi:10.1016/j.jpdc.2016.03.012.

[77] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 145–157, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164070.

[78] S. Knight, B. Maubert, and F. Schwarzentruber. Reasoning about knowledge and messages in asynchronous multi-agent systems. *Mathematical Structures in Computer Science*, pages 1–42, 2017. doi:10.1017/S0960129517000214.

[79] Dmitry Kozlov. *Combinatorial Algebraic Topology*. Springer, 2007. doi:10.1007/978-3-540-71962-5.

[80] Dmitry N. Kozlov. Combinatorial topology of the standard chromatic subdivision and weak symmetry breaking for 6 processes. *CoRR*, abs/1506.03944, 2015. URL: http://arxiv.org/abs/1506.03944, arXiv:1506.03944.

[81] Sanjeevi Krishnan. A convenient category of locally preordered spaces. *Applied Categorical Structures*, 17(5):445–466, 2009. doi:10.1007/s10485-008-9140-9.

[82] Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 387–396, 2018. URL: https://dl.acm.org/citation.cfm?id=3212765.

[83] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[84] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983. doi:10.1145/69624.357207.

[85] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.

[86] Jérémy Ledent and Samuel Mimram. A sound foundation for the topological approach to task solvability. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands.*, pages 34:1–34:15, 2019. doi:10.4230/LIPIcs.CONCUR.2019.34.

[87] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[88] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000. doi:10.1137/S0097539798335766.

[89] Alessio Lomuscio and Mark Ryan. On the relation between interpreted systems and kripke models. In Wayne Wobcke, Maurice Pagnucco, and Chengqi Zhang, editors, *Agents and Multi-Agent Systems Formalisms, Methodologies, and Applications*, pages 46–59, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[90] Carsten Lutz. Complexity and succinctness of public announcement logic. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*, pages 137–143, 2006. doi:10.1145/1160633.1160657.

[91] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7751.

[92] Paul-André Melliès and Samuel Mimram. Asynchronous games: innocence without alternation. In *International Conference on Concurrency Theory*, pages 395–411. Springer, 2007.

[93] Paul-André Melliès and Léo Stefanesco. A game semantics of concurrent separation logic. *Electr. Notes Theor. Comput. Sci.*, 336:241–256, 2018. doi:10.1016/j.entcs.2018.03.026.

[94] J. Misra. Axioms for memory access in asynchronous hardware systems. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 96–110. Springer Berlin Heidelberg, 1985.

[95] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Press, 1989.

[96] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

[97] Yoram Moses. Relating knowledge and coordinated action: The knowledge of preconditions principle. In *TARK*, pages 231–245. EPTCS, 2015. doi:10.4204/EPTCS.215.17.

[98] Leopoldo Nachbin. *Topology and order*. Van Nostrand mathematical studies. Van Nostrand, 1965.

[99] Gil Neiger. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

[100] Mogens Nielsen. Models for concurrency. In *International Symposium on Mathematical Foundations of Computer Science*, pages 43–46. Springer, 1991.

[101] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[102] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

[103] Timothy Porter. Geometric aspects of multiagent systems. *Electr. Notes Theor. Comput. Sci.*, 81:73–98, 2003. doi:10.1016/S1571-0661(04)80837-5.

[104] Timothy Porter. Interpreted systems and kripke models for multiagent systems from a categorical perspective. *Theoretical Computer Science*, 323(1):235 – 266, 2004. doi:10.1016/j.tcs.2004.04.005.

[105] Vaughan R. Pratt. Modeling concurrency with geometry. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 311–322, 1991. doi:10.1145/99583.99625.

[106] Y. Moses R. Fagin, J. Halpern and M. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[107] S. Rajsbaum. Iterated shared memory models. In *LATIN*, volume 6034 of *LNCS*, pages 407–416. Springer, 2010. doi:10.1007/978-3-642-12200-2_36.

[108] Martin Raussen. Invariants of directed spaces. *Applied Categorical Structures*, 15(4):355–386, 2007. doi:10.1007/s10485-007-9085-4.

[109] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO Conference*, pages 314–321, 1997.

[110] Rasmus Rendsvig and John Symons. Epistemic logic. In *The Stanford Encyclopedia of Philosophy,* see https://plato.stanford.edu/archives/sum2019/entries/logic-epistemic/. Metaphysics Research Lab, Stanford University, 2019.

[111] Joshua Sack. Logic for update products and steps into the past. *Ann. Pure Appl. Logic*, 161(12):1431–1461, 2010. doi:10.1016/j.apal.2010.04.011.

[112] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 101–110. ACM, 1993. doi:10.1145/167088.167122.

[113] Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *Distributed Computing - 30th International Symposium, DISC 2016. Proceedings*, pages 428–441, 2016. doi:10.1007/978-3-662-53426-7_31.

[114] Dana Scott and C. Strachey. Towards a mathematical semantics for computer languages. *Proceedings of the Symposium on Computers and Automata*, 21, 01 1971.

[115] G. Smith, K. Winter, and R. J. Colvin. A sound and complete definition of linearizability on weak memory models. *ArXiv e-prints*, February 2018. arXiv:1802.04954.

[116] Hans van Ditmarsch. Asynchronous announcements. *CoRR*, abs/1705.03392, 2017. URL: http://arxiv.org/abs/1705.03392, arXiv:1705.03392.

[117] Hans P. van Ditmarsch, Wiebe van der Hoek, and Barteld P. Kooi. Dynamic epistemic logic with assignment. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 141–148, 2005. doi:10.1145/1082473.1082495.

[118] Rob van Glabbeek. Bisimulation semantics for higher dimensional automata. Technical report, Stanford University, 1991. URL: http://theory.stanford.edu/~rvg/hda.

[119] Krzysztof Ziemianski. Spaces of directed paths on pre-cubical sets. *Appl. Algebra Eng. Commun. Comput.*, 28(6):497–525, 2017. doi:10.1007/s00200-017-0316-0.

Titre : Sémantiques Géométriques pour la Calculabilité Asynchrone

Mots clés : Sémantique, protocoles tolérants aux pannes, topologie, logique épistémique

Résumé : Le domaine des protocoles tolérants aux pannes étudie quelles tâches concurrentes sont résolubles dans différents modèles de calcul avec pannes. Des outils mathématiques basés sur la topologie combinatoire ont été développés depuis les années 1990 pour aborder ces questions. Dans ce cadre, la tâche que l'on veut résoudre, et le protocole auquel on fait appel, sont modélisés par des complexes simpliciaux chromatiques. On définit qu'un protocole résout une tâche lorsqu'il existe une certaine application simpliciale entre ces complexes.
Dans cette thèse, on étudie ces méthodes géométriques du point de vue de la sémantique. Le premier objectif est de fonder cette définition abstraite de résolution d'une tâche sur une autre plus concrète, basée sur des entrelacements de traces d'exécution. On examine diverses notions de spécifications pour les objets concurrents, afin de définir un cadre général pour la résolution de tâches par des objets partagés. On montre ensuite comment extraire de ce cadre la définition topologique de résolubilité de tâches.
Dans la deuxième partie de la thèse, on prouve que les complexes simpliciaux chromatiques peuvent être utilisés pour évaluer des formules de logique épistémique. Cela permet d'interpréter les preuves topologiques d'impossibilité en fonction de la quantité de connaissances à acquérir pour résoudre une tâche.
Enfin, on présente quelques liens préliminaires avec la sémantique dirigée pour les programmes concurrents. On montre comment la subdivision chromatique d'un simplexe peut être retrouvée en considérant des notions combinatoires de chemins dirigés.

Title: Geometric Semantics for Asynchronous Computability

Keywords: Semantics, fault-tolerant protocols, topology, epistemic logic

Abstract: The field of fault-tolerant protocols studies which concurrent tasks are solvable in various computational models where processes may crash. To answer these questions, powerful mathematical tools based on combinatorial topology have been developed since the 1990's. In this approach, the task that we want to solve, and the protocol that we use to solve it, are both modeled using chromatic simplicial complexes. By definition, a protocol solves a task when there exists a particular simplicial map between those complexes.
In this thesis we study these geometric methods from the point of view of semantics. Our first goal is to ground this abstract definition of task solvability on a more concrete one, based on interleavings of execution traces. We investigate various notions of specification for concurrent objects, in order to define a general setting for solving concurrent tasks using shared objects. We then show how the topological definition of task solvability can be derived from it.
In the second part of the thesis, we show that chromatic simplicial complexes can actually be used to interpret epistemic logic formulas. This allows us to understand the topological proofs of task unsolvability in terms of the amount of knowledge that the processes should acquire in order to solve a task.
Finally, we present a few preliminary links with the directed space semantics for concurrent programs. We show how chromatic subdivisions of a simplex can be recovered by considering combinatorial notions of directed paths.