Università degli Studi di Ferrara

Arenberg Doctoral School
Faculty of Engineering Science
Department of Mechanical Engineering

Dottorato di ricerca in Scienze dell'Ingegneria
Tesi in co-tutela

Dissertation presented in partial
fulfillment of the requirements for the degree of
Doctor in Engineering Science

CICLO XXVI
COORDINATOR: Prof. Stefano Trillo

Online Coordination and Composition of Robotic Skills: Formal Models for Context-aware Task Scheduling

Settore Scientifico Disciplinare: ING-INF/04

Dottorando - Ph.D. Candidate
Dott. Scioni Enea

Tutore - Supervisor UniFE
Prof. Bonfè Marcello

Tutore - Supervisor KU Leuven
Prof. Bruyninckx Herman

Anni 2011/2013

Preface

. . . and so it comes the time to write the most (likely) read part of a PhD dissertation! If the reader is brave enough, and he/she is looking for some technical insights about this work, the author suggests to avoid these rambling words and to jump to the Chapter 1: it is author's hope that he/she will find some answers to the questions that the author left behind (see Chapter 7). However, this dissertation is the outcome of a story with many actors, and if the reader is eager to know such a story, some details are provided below.

As many kids like me, to the question *"what will you be?"* my answer was *"an astronaut"*, but I immediately dropped the idea as soon as they told me that excellent math skills are required for that. However, I kept playing with my starship toys, and sometimes my father joined me. One evening, he was stealing top-secret information from my imaginary Moon station, thanks to a mysterious robot hidden in my base. That was the first time I ever heard about an intelligent and autonomous machine, and back then, none of us had any idea about the impact of such an episode had on me. Moreover, my father forgot to mention that being a roboticist requires some math skills too! I will always be grateful to my mother Cledes and my father Graziano to keep such a secret for so long, and to allow me to pursue any goal I had in mind. *Grazie Mamma, grazie Babbo.*

The years have passed, and I decided to pursue a technical degree instead of following classical studies during the high school, with a focus on *automation*. Along this period, I started developing some insane interests about programming and computers in general: initially, everyone (including me) thought that was merely related to the time spent playing videogames, but, obviously, it was not.

Five years later I decided to pursue a degree in Computer Science and Automation Engineering at the University of Ferrara. During both my bachelor and master career, Prof. Marcello Bonfé taught me the fundamentals of embedded systems and control. A future career in robotics was not clear by then, until it was the time to choose a topic for my master thesis. Almost by chance I applied for an Erasmus scholarship, and actually Sweden was my first choice. However, as soon as I hit Prof. Herman Bruyninckx web-page, I changed my mind and my destination, provoking some headaches to the university administrations[1], since the position at KU Leuven was for a student in electronic engineering, and obviously not at the mechanical department. Once again, I had no idea that this was just the beginning: I planned to stay abroad only six months, but it turned out to be six years! (Sorry Mom).

After my Master thesis, Marcello suggested me an opportunity to pursue a Ph.D at University of Ferrara, and because of my previous rewarding experience at KU Leuven, as well as my strong interest on software design applied to Robotics, I expressed my wish to carry on a collaboration between Ferrara and Leuven, at least in the context of my research. After few months, my wishes got shaped in a joint agreement between the two Universities, and this was a *one time in life opportunity*.

It has been a long, bumpy road, with uncertainties, weekends in the office, nights in the lab, and travels all around the globe (Seattle, Tokyo, Chicago). Many directions and detours have been taken, but full of lessons learnt and experiences, which goes from theoretical issues down to practical troubles. I must admit that it has not been easy, but along the third year my research got a concrete shape, which happened to be in-line with the KU Leuven role in the EU-FP7 *RoboHow.cog* project.

From this short resume of my life, it is clear to me that a massive amount of gratitude goes to Herman and Marcello to give me an awesome opportunity, a large number of critical and constructive discussions flavored with a coffee, as well as the freedom to search my own answers.

I would like to express my gratitude to the members of the examination board, who spent their time for gathering in Ferrara: Prof. Joris De Schutter, Prof. Cristian Secchi and Prof. Lorenzo Marconi.

This work would have not been possible without the help and the expertise of two amazing colleagues: Gianni Borghesan, who strongly contributed on my research, and for which I give a special thank regarding Chapter 3, result of a joint effort in the context of *RoboHow.cog*; Markus Klotzbücher, my software mentor, for changing my way to approach to software development.

Five years are long and I have met amazing people who helped me of growing up as a scientist and as a human being; I would like to thank, in a casual order: Tinne De Laet for her help in the early stage of my academic experience in

[1] I kept being annoying for several years after that.

Leuven; Nico Hübel for his advices during long coffe breaks; Erwin Aertbeliën and our everlasting discussions; Enrico Di Lello, who has welcomed me in Leuven; Niccolò Tosi and his friendship; (Domi)Nick Vanthienen and the PR2 adventures; Lin Zhang and the "chicken legs sundays"; *Dottor* Wilm Decré; Koen Buys and his hospitality; Azamat Shakhimardanov and Sebastian Blumenthal, remote colleagues from Bonn; Maxim, Jonas, Bart, Kevin for tolerating my rants as an officemate in Leuven, and Nicola Preda for my rants in Ferrara. Moreover, I believe that my fish of plastic deserves an acknowledgement, as well as the jesus who reminded me to keep up the writing.

There is a bunch of people that really make this experience enjoyable, especially during my staying in Leuven. The list is very long, but I feel like to spend few words anyway: Lucie and the late afternoon tea; Alireza, Sjoerd, Philip, Marco and all the Happy Hour people, reminding me that weekends exist; Inés *y sus bailarinas*; Mirian *y el timbre 175*; Matteo, the homebrewing and plenty of friendly yeasts. I am grateful to Simona and our hate-love-hate relationship, sometimes packaged in tupperware. A very special thank goes to Sergio (Frodo), colleague and mate of many adventures with plenty laughter and few tears: my staying in Leuven would not be the same without your support.

I would like to thank who, after years abroad, always welcome me like I never even left: Enrico (Savio) Savioli, Nicola (Tino) Fantini, Naike Tallevi and Alessandro (Raso) Rasini.

My deep gratitude goes to my grandfather Laerte, from whom I have inherited my stubbornness, and to my grandmother Venerina, who is still surprised that I am not going hungry in Belgium: *grazie nonno, grazie nonna!*

Maria, *gracias por esperar por mi, por todo tu apoyo y por recordarme que no siempre hay que mantener los pies en el suelo, que encima de un arco iris se está muy bien.*

Ferrara, Aprile 2016,
Enea Scioni

Acknowledgements

This dissertation is presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science and Dottore di Ricerca, in the context of a joint PhD agreement and in close collaboration between *University of Ferrara* (Italy) and *KU Leuven* (Belgium). The author is very grateful to both institutions for the great opportunity of carrying out his research in this international environment.

This research was funded by the Italian Ministry of Education, University and Research (MIUR, *Ministero dell'Istruzione, dell'Università e della Ricerca*), through a PhD scholarship *"Borsa di Dottorato di Ricerca"* and by the European Commission through the FP7 project *RoboHow.cog* (FP7-ICT-288533).

Abstract

This research aims to bridge the gap between symbolic plans and executable, constraint-based robot task specifications by means of offline skill programming, and online skill scheduling, refinement and adaptation. Symbolic plans are *discrete* sequences of actions that must be performed by a robot to achieve an intended change in the environment. Such actions are not directly executable, so a refinement process adds *context-dependent* knowledge (about the robot, its tools, and the environment) required to generate the robot's *continuous* motion.

This work enhances this refinement process, providing a declarative description of those situations that lie in both discrete and continuous domains, and a late online decision making about the symbolic and continuous representations of an action. For example, a multi-arm robot may have symbolic plans to open a drawer by means of the right arm, the left arm, or both, and it can delay its decision about which plan to execute until it knows all properties of the drawer and its environment. In the same vein, grasping the drawer's handle requires the robotic gripper to be opened, and while that action can start anytime during the approach motion, the progress of both actions must be coordinated online, to avoid the robot to hit the handle with a not yet sufficiently opened gripper. The state-of-the-art solution is typically to first open the gripper, and only then start the approach motion; this work optimises the plan execution at runtime, by an execution engine that exploits the most current status of the robot capabilities. This behaviour is supported by formal models and *Domain Specific Languages*, and by mechanisms to specify, compose and coordinate robotic skills at runtime.

A first contribution is a formal skill model that represents both discrete and continuous aspects of an action. The scheduling of the execution of these skills is constrained by a set of dependencies that must be satisfied online, and that represent *logical conditions* on the *continuous state* of the robot-environment interaction; the formal representation of these dependencies is called the *Skill Dependency Graph* (*SDG*). Such *SDG* models are interpreted and realised at runtime by an execution engine called *Skill Dependency Graph Executive.* This dissertation prefers *declarative* and *formal* representations instead of *procedural* and *code-based* approaches: to this end, the *micro Skill Dependency Language* (*uSDL*) is proposed as a minimal set of declarative rules behind *SDG* models.

The second contribution of this dissertation is a novel constraint-based task specification based on formal, declarative geometric relationships to represent motion constraints; that is, relationships (*e.g.*, distances and angles) between pairs of geometric entities (such as points, lines and planes). At runtime, these formal expressions are evaluated in the actual robot-environment context to generate explicit and instantaneous motion set-points, optimising the robot limitations (*e.g.*, joint limits) and redundancies. Moreover, this solution decouples the motion specification from the underlying numerical solver; in this context, constraint-based optimal control strategies are preferred to constraint-based motion planning to provide reactiveness and to support specifications that describe physical interactions.

The third contribution is to integrate the previous two, with the explicit aim to support *generalization* over skill models. Geometric entities are modelled as part of a geometric item description, whether it is a physical object, a virtual object or a tool (*i.e.*, a controlled object that adds further capabilities to the robot). This modelling effort generalises the skill specification to a *class* of cases, where the objects involved are described with semantically equivalent and more generic geometric entities. For example, the skill that performs the grasping of a drawer's handle conforms to the same generic skill as the grasping of a glass, since both handle and glass are approximated as a cylindrical envelope. This generalisation leads to *skill prototypes*, which are not executable *as is*, but require additional *context-dependent* knowledge: the output of this refinement is a executable *skill instance.* Complex skills are created by *composition* of multiple skill prototypes, preferred to *inherit* from a single skill; several experimental examples are provided in this work.

Another contribution is the realisation of a *Just-in-Time* strategy, to postpone decisions until they are really needed, so that the task specifications are composed with the latest information available at runtime.

A final contribution proposes *hierarchical hypergraphs* and a meta model called "*NPC4*" as formal representation of the structural aspects of graph-based models (including the *SDG*). Such a strict separation between *structure* and *behaviour* enables the reuse of infrastructure code, and simplifies the definition of new Domain Specific Languages.

Sommario

Lo scopo di questa tesi di dottorato consiste nel trasformare piani di tipo simbolico in specifiche di movimento basate su vincoli ed eseguibili da sistemi robotici autonomi, attraverso la programmazione di *skill* di movimento, nonchè la loro schedulazione e adattamento a tempo di esecuzione. I piani di tipo simbolico sono sequenze ordinate di azioni *discrete* eseguite da un robot atte ad ottenere un desiderato cambiamento sull'ambiente circostante. Le azioni, in quanto simboliche, non sono direttamente eseguibili, poiché esse non contengono dettagli informativi che dipendono strettamente dal *contesto di esecuzione* dell'azione stessa (*e.g.*, informazioni sulle caratteristiche del robot, degli strumenti del quale è equipaggiato, nonchè informazioni sull'ambiente circostante); per rendere eseguibili le azioni è quindi necessario un processo di conversione di ciascuna azione simbolica in una *skill* che definisca come eseguire il movimento *continuo* del robot.

Questa tesi propone una metodologia per formalizzare la suddetta trasformazione, fornendo una descrizione dichiarativa di quelle situazioni che esibiscono una natura sia discreta che continua e ritardando tale processo laddove possibile. Per esempio, un robot umanoide può aprire un cassetto per mezzo del braccio destro, sinistro o entrambi: la scelta della strategia di apertura dev'essere ritardata fintanto che le informazioni siano insufficienti per una decisione adeguata. Inoltre, afferrare la maniglia di un cassetto richiede che la pinza robotica sia aperta: quest'ultima azione può essere eseguita in qualsiasi momento durante il movimento di avvicinamento alla maniglia, ma evitando di urtare quest'ultima nel caso in cui la pinza non sia sufficientemente aperta. Una soluzione semplicistica è quella di eseguire tali azioni in ordine strettamente sequenziale: questa tesi propone un'alternativa che ottimizza l'esecuzione del piano online, attraverso un motore d'esecuzione che sfrutta le caratteristiche del robot e le adatta al contesto di esecuzione.

Un primo contributo della tesi è la definizione di un modello formale di *skill* robotica che permetta di rappresentare un'azione in modo completo, considerandone gli aspetti nel dominio sia discreto che continuo. La schedulazione dell'esecuzione di queste skill è vincolata ad un insieme di dipendenze da soddisfare a tempo di esecuzione; tali dipendenze rappresentano *condizioni logiche* relative alle interazioni tra robot e ambiente, le quali appartengono al dominio *continuo*. La rappresentazione formale di queste dipendenze definisce un modello chiamato *Skill Dependency Graph* (*SDG*), che viene interpretato e realizzato a tempo di esecuzione da un *Skill Dependency Graph Executive* (*SDG*-E). Preferendo una descrizione *dichiarativa* e *formale* rispetto ad approcci *procedurali*, questa tesi propone inoltre il linguaggio *micro Skill Dependency Language* (*uSDL*), come un insieme minimale di regole dichiarative che definiscono le relazioni in un modello *SDG*.

Come secondo contributo, la tesi descrive un'innovativa specifica di movimento per mezzo di relazioni geometriche che ne rappresentano i vincoli, ovvero relazioni tra coppie di entità geometriche (punti, linee e piani) come distanze ed angoli. A tempo di esecuzione, queste espressioni formali sono valutate nel contesto attuale allo scopo di generare setpoints istantanei di movimento, ottimizzati rispetto ai limiti del robot (*e.g.*, limiti di giunto) e ridondanza cinematica. Inoltre, questa soluzione separa la specifica di movimento dal risolutore numerico adottato; in tal senso, strategie di controllo ottimo su vincoli sono preferite alla pianificazione del movimento su vincoli, in quanto le prime supportano specifiche che includono interazioni fisiche o che implementano azioni di controllo reattive.

Un terzo contributo è derivato dall'integrazione dei due precedenti, avente lo scopo di *generalizzare* quelle componenti che formano un modello di skill programmato manualmente. Le entità geometriche discusse in precedenza sono modellate come parte di generici elementi geometrici, che siano oggetti fisici, oggetti virtuali o strumenti del robot stesso. Questo approccio permette di classificare le varie specifiche di skill robotiche, laddove gli oggetti coinvolti nella specifica stessa siano semanticamente equivalenti. Per esempio, la skill che realizza il movimento per afferrare una maniglia di un cassetto è conforme alla stessa skill generica che afferra la maniglia di una porta o un bicchiere, in quanto questi oggetti possono essere approssimati come inviluppo di un cilindro. Una skill cosí generalizzata è detta *prototipo*: skill prototipali non sono direttamente eseguibili, ma sono arricchite da un processo di trasformazione che provvede informazioni *dipendenti dal contesto* di esecuzione. Il risultato di tale processo converte il prototipo in un'istanza eseguibile in una skill di movimento. Skill complesse sono create per *composizione* di molteplici prototipi, piuttosto che applicando una strategia ad ereditarietà da una sola skill. Questo lavoro di tesi provvede diversi esempi di skill implementate e testate sperimentalmente.

Questa tesi contribuisce alla realizzazione di una strategia *Just-in-Time*, con lo scopo di ritardare ogni decisione fintanto che quest'ultime siano effettivamente necessarie per la realizzazione del movimento. Grazie a questa soluzione, la specifica di movimento è composta a tempo di esecuzione considerando le ultime informazioni disponibili.

Infine, questa tesi propone ipergrafi gerarchici e un meta modello chiamato "*NPC4*" per rappresentare formalmente aspetti strutturali di modelli basati su grafi (incluso il modello *SDG*). La separazione tra *struttura* e *funzionalità* permette un miglior riutilizzo di quelle porzioni di codice in comune, inoltre ne semplifica la definizione di nuovi linguaggi specifici di dominio.

Beknopte samenvatting

Dit onderzoek probeert een brug te slaan tussen symbolische plans enerzijds, en uitvoerbare beperkingsgebaseerde taakbeschrijvingen anderzijds, via de offline programmatie van vaardigheden, en het online schakelen van hun uitvoering, hun verdere verfijning, en hun aanpassing. Symbolische plans zijn *discrete* actiesequenties die de robot moet uitvoeren om een beoogde verandering in de wereld te realiseren. Zulke acties worden pas uitvoerbaar, in de vorm van *continue* robotbewegingen, als er *contextafhankelijke* kennis is toegevoegd over de robot en zijn gereedschappen, en over de omgeving.

Dit werk breidt dit verfijningsproces uit, door declaratieve beschrijvingen te leveren van situaties die zowel in het symbolische als continue domein liggen, en door late online beslissingen te nemen over de symbolische en continue voorstelling van acties. Zo kan de robot bijvoorbeeld symbolische plans hebben om een schuif open te trekken met de rechterarm, met de linkerarm, of met beide, en hij kan zijn beslissing uitstellen over welk plan hij effectief zal uitvoeren totdat hij weet heeft van alle relevante eigenschappen van de schuif en de omgeving. Op dezelfde wijze vereist het grijpen van het handvat aan de schuif dat de grijper van de robot openstaat; die openingsactie kan gelijk wanneer starten gedurende de naderingsbeweging naar de schuif toe, maar de vooruitgang van beide moet wel gecoördineerd worden gedurende de uitvoering, opdat de robot niet met een onvoldoende geopende grijper het schuifhandvat zou raken. State-of-the-art aanpakken openen typisch eerst de grijper, en starten pas dan met de naderingsbeweging; dit onderzoek optimiseert de uitvoering van taken online, door een “execution engine” dat gebruik kan maken van de meest actuele informatie over de robot. Deze aanpak is gebouwd met formele modellen, zogenaamde *Domain Specific Languages*, en met mechanismen om robotvaardigheden online te specificeren, samen te stellen en te coördineren.

Een eerste bijdrage is een formeel model van vaardigheden dat zowel de symbolische als de continue aspecten van acties voorstelt. De schakeling van de uitvoering van deze vaardigheden is onderhevig aan een verzameling van

beperkingen en afhankelijkheden die online moeten voldaan zijn, en die *logische voorwaarden* voorstellen op de *continue toestand* van de interactie tussen robot en omgeving; de formele voorstelling van deze afhankelijkheden heet de *Skill Dependency Graph* (*SDG*). Zulke *SDG* modellen worden online geïnterpreteerd en gerealiseerd door een execution engine, met de naam *Skill Dependency Graph Executive*. Deze thesis geeft de voorkeur aan *declaratieve* en *formele* representaties boven *procedurele* en *codegebaseerde* aanpakken. Met dit doel is de *micro Skill Dependency Language* (*uSDL*) ontwikkeld, als een minimale verzameling van de declaratieve regels achter de *SDG* modellen.

De tweede bijdrage van de thesis is een nieuwsoortige beperkingsgebaseerde taakspecificatie op basis van formele, declaratieve geometrische relaties die bewegingsbeperkingen voorstellen. Dat wil zeggen, relaties zoals afstanden en hoeken, tussen meetkundige primitiven zoals punten, lijnen en vlakken. Deze formele uitdrukkingen worden online geëvalueerd in de context van de huidige robotomgeving, om zo de expliciete en ogenblikkelijke streefwaarden te genereren voor de beweging van de robot, tevens rekening houdende met zijn beperkingen (zoals gewrichtslimieten) en redundanties. Bovendien ontkoppelt deze aanpak de bewegingsspecificatie van de onderliggende numerieke oplossingsmethode; beperkingsgebaseerde optimale controle-strategieën zijn te verkiezen boven beperkingsgebaseerde bewegingplanning, om hogere reactiviteit mogelijk te maken, in allerhande soorten fysische interacties.

De derde bijdrage is gericht op de integratie van de eerste twee, met het oog op het kunnen realiseren van *veralgemeningen* van vaardigheidsmodellen. Geometrische entiteiten worden gemodelleerd als een onderdeel van de beschrijving van fysische objecten of virtuele objecten, of van werktuigen (dwz., objecten met een eigen controller die de mogelijkheden van de robot uitbreiden). Deze modelleerinspanning veralgemeent de specificatie van vaardigheden tot *klasses*, waarin de betrokken objecten voorgesteld worden met semantisch equivalente en meer algemene geometrische entiteiten. Zo voldoet bijvoorbeeld de vaardigheid om een schuifhandvat te grijpen aan dezelfde veralgemeende vaardigheid als de vaardigheid om een glas te grijpen, omdat beide benaderd kunnen worden als cylindrische primitieven. Op deze manier krijgt men *vaardighedenprototypes*, die niet op zichzelf uitvoerbaar zijn, maar hiervoor verder verfijnde *context-afhankelijke* kennis nodig hebben: zo'n meer verfijnde uitvoer is een *instantie* van een vaardigheidsprototype. De *compositie* van zulke vaardigheden levert steeds complexere vaardigheden op; en compositie krijgt hierbij de voorkeur op overerving. Verscheiden experimentele voorbeelden worden in de thesis uitgewerkt.

Een andere bijdrage van de thesis is de realisatie van een *Just-in-Time* strategie, om beslissingen uit te stellen tot wanneer ze echt nodig zijn; dit maakt het mogelijk om taakspecificaties samen te stellen op basis van de allerlaatste

informatie die online beschikbaar is.

De laatste bijdrage voert *hierarchische hypergrafen* in, met een meta model met de naam "*NPC4*", als formele voorstelling van de structurele aspecten van grafengebaseerde modellen (met inbegrip van de *SDG*). Zulke stricte scheiding tussen *structuur* en *gedrag* maakt het mogelijk om basiscode te hergebruiken, en vereenvoudigt de definitie van nieuwe Domain Specific Languages.

Abbreviations

3T-A	: Three Tiered Architecture
AI	: Artificial Intelligence
API	: Application Program Interface
AST	: Abstract Syntax Tree
COP	: Constraint Optimization Problem
CRAM	: Cognitive Robot Abstract Machine
CPL	: CRAM Plan Language
CSP	: Constraint Satisfaction Problem
CTAMP	: Combined Task and Motion Planning
DSL	: Domain Specific Language
eTaSL	: expressiongraph-based TAsk Specification Language
FSM	: Finite State Machine
MDE	: Model-Driven Engineering
NPC4	: Node Port Connector Container Contains Connects (Language)
iTaSC	: instantaneous Task Specification using Constraints
JSON	: JavaScript Object Notation
LTL	: Linear Temporal Logic
QoS	: Quality of Service
QP	: Quadratic Programming
qpOASES	: Quadratic Programming with Online Active-Set Strategy
PDDL	: Planning Domain Definition Language
PRS	: Procedural Reasoning Systems
RAP	: Reactive Action Packages
ROS	: Robot Operating System
RRT	: Rapidly exploring Random Tree
rFSM	: Reduced Finite State Machine
SDG	: Skill Dependency Graph
SDG-E	: Skill Dependency Graph Executive
SoT	: Stack of Tasks

SQP	: Sequential Quadratic Programming
SESD	: Skill Execution Status Diagram
STRIPS	: Stanford Research Institute Problem Solver
SWBC	: Stanford Whole Body Control
TCA	: Task Control Architecture
TDL	: Task Description Language
TFF	: Task Frame Formalism
uSDL	: micro Skill Dependency Language

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since its first announcement in 2011, the forthcoming *Industry 4.0* (the fourth industrial revolution) has drawn the attention of manufacturing industries to the adoption of new technologies, such as *Cloud Computing*, the *Internet of Things* and *Smart Factories*. One of the driving factors for such a strategic innovation involves different branches of the Robotics research, from *Artificial Intelligence* (AI) to *control* of autonomous robotic systems. An expected scenario promises to increase the overall productivity (and to reduce the related production costs) through a novel industrial environment, where machines are not hidden behind cages, but they work autonomously and co-operate with human operators. These innovations are also inspired by recent results in numerous research projects funded by the 7th European Framework Programme, that even go further by introducing autonomous robots in human working and living environments; examples of those projects are *GeRT* [60], *Rosetta* [130] and *RoboEarth* [164]. The *RoboHow.cog* project [128] investigated the role of the knowledge acquired from instructions in the World Wide Web and from previous human demonstrations to generate executable plans. One part of this process concerns the development of autonomous robotic *skills*, such that robots can be easily programmed by instructions which are as close as possible to a natural language. At run-time, the robot adapts the instructions received to the *context* of the execution, by exploiting the previously acquired knowledge.

However, it is inherent in human nature to describe a *situation* as a linear sequence of actions, whether they represent a plan to be executed or a fact that happened previously. This limitation is still reflected in many robot programs, which are affected by hidden assumptions over the context of the execution. The "non-linear nature of plans" is already known in AI literature [134, 105],

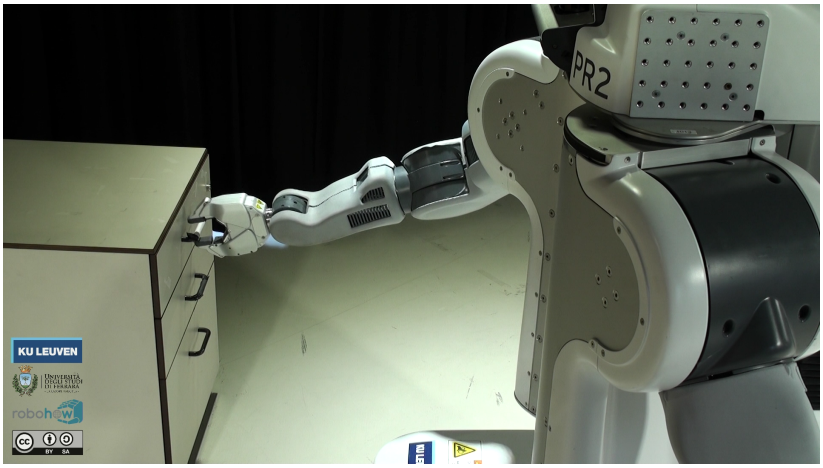
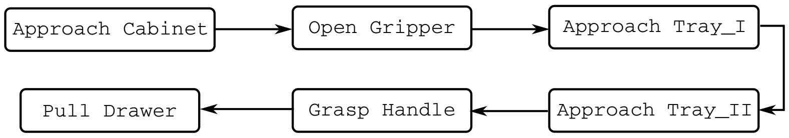


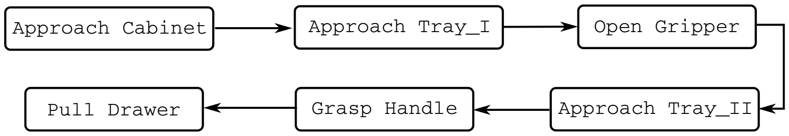
Figure 1.1: An autonomous robot opening a drawer (frame from an experiment discussed in Chapter 2).

which provides a first definition of *total-order* and *partial-order* plans: where a plan is an ordered set of *actions*, that plan is *totally-ordered* if the order relationship is known for each action, while it is *partially-ordered* if only some actions are ordered between each other. Thus, a partial-order plan is *non-linear*, and a *linearisation process* can produce several total-order plans from it. This *linearisation process*, sometimes called *"refinement"*, is possible when plan uncertainties are solved by *context-dependent* information over the plan execution.

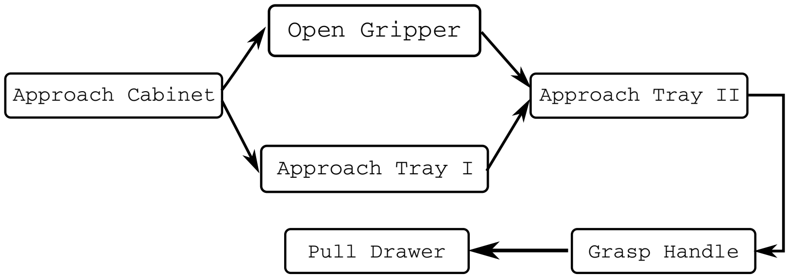
An intuitive example is provided by the plans in Figure 1.2, which describe the symbolic actions that a robot should execute to perform the opening of a drawer (Figure 1.1). The possibility to concurrently execute multiple actions depends on their implementations; as an initial assumption, concurrent action execution is not possible, but this assumption can be removed as discussed in the next paragraph. Considering the total-order plan in Figure 1.2a: firstly, the robot approaches to the cabinet; secondly it opens its gripper; then it approaches to the tray, bringing the gripper to a sufficient distance to grasp the handle; finally, the handle is grasped and a pulling action is applied to the drawer. The approach to the tray phase can be implemented by a couple of atomic actions, `Approach Tray I` and `Approach Tray II`. The former brings the gripper close to the handle, while the latter implements a local movement that brings the gripper in the grasping pose, thus requiring the gripper to be opened in advance. As a consequence, the order between the action `Open Gripper` and `Approach Tray I` is not defined prior to execution, while `Approach Tray II` exhibits a



(a) A total-order plan for opening a drawer.



(b) An alternative total-order plan for opening a drawer.



(c) A partial-order plan for opening a drawer. A linearisation process can generate one of the two total-order plan above (Figure 1.2a and Figure 1.2a).

Figure 1.2: A partial-order plan that describe the opening of a drawer (Figure 1.2c), and two linearisation alternatives (Figures 1.2a and 1.2b).

dependency on the gripper status. Figure 1.2b shows an alternative total-order plan, where the order of `Open Gripper` and `Approach Tray I` are reversed with respect to the previous plan in Figure 1.2a. Figure 1.2c illustrates a partial-order plan that describes both situations: a linearisation process leads to one of the previous total-order plan. Resuming, this example shows that there exists a need of declarative specifications over procedural descriptions, in which the order is not explicitly defined, but rather a set of relationships determine the execution order of the planned actions from some knowledge not available at planning time.

However, the above-mentioned plans are not efficient due to the initial assumption of executing only one action per time. It is obvious that the resulting behaviour will benefit from concurrent scheduling, *e.g.*, executing both `Open Gripper` and `Approach Tray I` at the same time. Other actions can be part of the same *online activity*, but it is not trivial to identify those prior to execution: it depends on how the actions are implemented, but also on the *robot capabilities* and the kinematic redundancy available on the robot. For example, the robot in Figure 1.1 has two arms, and maybe one is better positioned than the other to execute the task; even both arms can be used. Furthermore, the environment in which the robot operates is dynamic, and this prevents any form of early optimisation. Therefore, optimisation on the motion execution is delegated to run-time decisions to be made by control algorithms that implement the intended behaviour of the actions.

Opening a drawer and manipulating other mechanisms such as doors are well-known control problems already studied in the past; some examples are [73, 124, 95, 77, 76]. Even if they are efficient, these approaches only consider the whole problem in the *continuous domain*, or at most few hand-written discretisation steps are employed, often encoded into a Finite State Machine (FSM). This *procedural approach* implies multiple hidden assumptions on the feasibility of the plan, *e.g.*, only a nominal execution is considered, as well as on the assumptions over the chosen tuning parameters that feed models and control algorithms. In addition, some control laws are employable under the assumption that the robot is equipped with a specific tool or sensor (*e.g.*, force sensing capability), which must be verified. As a consequence, the *continuous* control of the robotic platform is often optimised, but only with respect to a situation that is known offline already: composition, flexibility and reusability, in a broader context and with run-time inputs and decisions, are not provided.

How to bridge the gap between symbolic plans and continuous motion control is a very open research question, which is not only due to technical difficulties, but also to the heterogeneous background that composes the Robotic Community. As a matter of fact, such an integration requires a deep understanding of both the AI and Control literature, which often use different terms to express the same concepts. This leads to numerous omissions over the assumptions taken, sometimes as an unconscious fact, sometimes as a conscious lack of interest from the researcher. Recent position papers [61, 71] point out the need to renew research on the synergy between planning and acting, and this dissertation contributes with a concrete research question:
given a symbolic plan, how to schedule, compose and optimise both discrete and continuous aspects of its execution, while still preserving logical consistency and the reactiveness over non-nominal situations?

1.1 Problem Statement

The previous narrative scenario shows, together with other practical experiences on software development for Robotics, a set of challenges and research questions that this dissertation addresses.

Q1: Which are the hidden assumptions that a programmer makes when he/she is building a robotic application, whether it is an high-level program or the implementation of a single action? How to model explicitly those assumptions?
Plans often represent only *nominal* situations, that is a sequence of actions to perform, valid only if the environment reacts as expected. Most of the applications run successfully because the context of their execution is known *a priori*, or even controlled. In the same vein, a control strategy that implements an action suits to a particular scenario, where a robot that has some capabilities interacts with well-defined objects. Offline optimisation and a proper tuning of the control parameters is possible due to the assumptions made in advance. This research aims to model these *context-dependent* assumptions, with the objective to exploit those for run-time optimisation and adaptation. Therefore, this raises the following question:
Q2: How to compose and to schedule multiple actions concurrently, thus generating an optimised execution with respect to the robot capabilities and the context of the execution?
To this end, it is necessary to bring formal knowledge into action descriptions, defining a motion specification that the robot must execute.
Q3: How to transform a symbolic action into a continuous motion specification?
Q4: Which model can describe an action in both discrete and continuous domains?
In the planning domain, the status of the execution of an action is monolithic, that is, an action can be executed, not executed or under execution. There is no information about the continuous execution of the action; as a consequence, it is not possible to reason and to react on the continuous execution. Moreover, an action may change the environment in many ways, and not all the effects of an execution are intended. An objective is to monitor the continuous behaviour performed by the execution of an action, and to react in case of conditions that diverge from the planned situation. Consequently, the following question arises:
Q5: Which conditions are relevant for the execution of an action? How to map the information monitored in the continuous domain to the discrete domain?
Q6: How to model a reaction to a violation of some conditions?
The implementation of an action should consider also non-nominal situations,

inducing a change in the motion specification. Yet another research question is **Q7: Which is a minimal set of declarative rules that describe a situation, thus creating a relationship between the execution of different actions?**
A plan can be described as a procedural program, as well as a composition of explicit logical constraints that must be satisfied. This dissertation investigates about the latter, such that the scheduling of the actions is adapted to the run-time conditions.
Q8: When must the context-dependent knowledge be available?
Most of the control frameworks require that the motion specification is completely defined at one specific stage of the application, sometimes at configuration time, sometimes offline. However, it is not necessary to define the whole plan at once, fulfilling information subject to changes, but to delay such a refinement until it is strictly necessary: formalise such an approach is part of the objectives of this work.

Q9: Can the software infrastructure be generalised to serve other purposes than the one for which it is designed?
This work addresses the above-mentioned research questions through formal models. However, software development covers an important role to support and to "activate" those models; during the development process, researchers and engineers are often focused on solving only the problems that they are addressing, while part of the software infrastructure may serve other purposes, even in different contexts. This regards not only the re-use of some functionalities, but also the structural aspect that represents and stores the data, and this dissertation aims to address also this facet.

As a whole, these research questions lead to elaborate a broader, but concrete objective, which is: **how to bridge a symbolic plan, which lies in the discrete domain, and constraint-based motion control, which lies in the continuous domain?**

1.2 Terminology

In literature, there is no common agreement about the definitions of *actions*, *skills* and *tasks*. Sometimes these terms are used interchangeably, sometimes they have a specific meaning, leading to a linguistic ambiguity. This is yet another symptom of the heterogeneous nature of the Robotic Community.

In Control, the term *action* refers to a *control* action, that is the output of a control strategy; in [162, 161] an action is an elementary motion primitive; in [61] an action is a symbolic world-transformation step that can be used

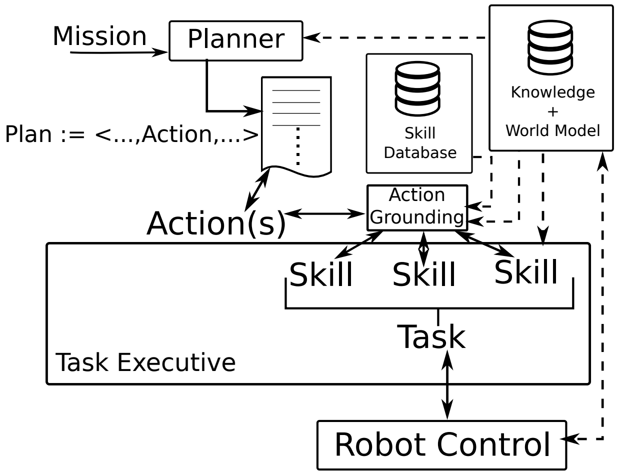


Figure 1.3: A generic functional architecture of a robotic system. From a mission, a planner constructs a feasible plan, taking into account the knowledge about the world representation and the agents available to accomplish the mission. The plan is a total or partial-order list of symbolic actions. The actions are transformed in skills instances from a given database of skills, taking into account environmental context and robot capabilities. A plan executive manages the run-time activation of the skill instances with respect to the relationship between them. The skills activated at run-time composes the overall task as the activity executed by the robot, realising the desired behaviour. Some (sub-)tasks involve perception capabilities of the robot, so that the information about the world representation is updated at run-time. In case of unforeseen situations, a new task can be re-composed or, in the worst case, recall a new planning phase from updated information.

to perform a certain task. A definition that suits all the above is the one of *acting* [71], that is an implementation of an on-line, closed-loop feedback function that processes streams of sensor stimuli to actuators commands in order to refine and control the achievement of an intended condition. According to this definition, an action requires a refinement mechanism to implement and instantiate a concrete motion; this refinement process involves further knowledge of both environment and robot capabilities, and the result of this process is a *grounded* skill, that is, the description of which actions should be performed, together with specifications about how the robot capabilities should be used, and how the run-time reaction should be to the environment context.

For the sake of clarity, the following terminology complies with the definitions given in [71, 61, 67, 120], but it is also influenced by the overall workflow that this dissertation proposes, briefly depicted in Figure 1.3, and by the adopted methodology described in Section 1.3.

Mission
A mission represents the overall intention of a robotic application. For example, “make a pizza”, “set the table for dinner” and “pour some water in a glass” are descriptions of missions expressed in natural language, describing a world transformation at a different level of abstraction (*e.g.*, “pour some water in a glass” can be part of “set the table for dinner”). In general, a mission does not specify the agent(s) responsible for performing those changes, and assigning a role to each agent is part of the mission planning problem.

Symbolic Plan
A symbolic plan is a situation[1] that an agent should perform, described by means of a total or partial-order list of symbolic actions to execute. A plan differs from a mission, since not only the world transformation are described, but also the set of actions that can be performed. A plan is a refined version of a mission, where the role of the agents is known; the latter can hold depending on the level of abstraction, *e.g.*, a robot can execute a plan to “pour some water in a glass”, but it could not be decided which arm(s) will perform such a world transformation. In literature, the statement that “a robot is performing a *task*” often refers to the execution of a plan.

Symbolic Action
A symbolic action lies in the symbolic domain and it represents a physical motion with the intention to achieve a world-transformation visible in the symbolic domain. A symbolic action can be composed of other actions; in this case, the action *abstracts* a symbolic plan. An action is executed by an agent, whether it is an autonomous robot or a human, and it can accidentally cause other world-transformations besides the intended one.

Symbolic Planning
Symbolic planning is a process that computes a symbolic plan that performs a given mission from a known initial state.

Motion Skill
A motion skill is a model that represents the implementation of an action that resides in the control domain. Other types of skills are *perception skills*; since this dissertation focuses on motion skills, the term “motion” is omitted. The skill model abstracts the information of a control strategy which is reusable in similar contexts, and it is executable if the knowledge about the context of the execution is fully available; in the latter case, the skill represents a concrete

[1]In AI, a situation is a feature-based representation of actions, that is, the description of the world changes by means of actions and their effects. A situation must not be confused with the state of the world. A situation can describe future intentions (*i.e.*, plan), but also not intended circumstances and occurred actions.

instance of a motion description. A skill *prototype* is an implementation of a generic action that is legal under a set of pre-defined assumptions about the context in which the action must be performed. Skill prototypes can be programmed by a skill developer, and they can be employed whenever the assumptions made on the environment context holds; however, they are not executable as is, but they require a further refinement process that involves run-time information. Alike symbolic actions, a skill can be composed of other skills, and the relationships between those define a concrete executable plan.

Ground a Symbolic Action to an Executable Motion
Grounding a symbolic action refers to the refinement process that transforms such an action into an executable motion description. In the scope of this dissertation, a symbolic action is grounded into an executable skill instance, and this transformation adds extra knowledge to the symbolic representation.

Context-awareness
Context-awareness is the capability of an automated system to select and adapt the grounding and the execution of an action with respect to the mission, the environment and the robot capabilities. Context-awareness includes the capacity of selecting the quantities to be monitored, as well as to provide a correct interpretation to the latter. From the given definition, skill programming plays a fundamental role for the fulfillment of a context-aware robotic system.

Skill Scheduling
Skill scheduling is a process that decides *when* a skill instance is executed at run-time, that is, the linearisation process that determines the execution order of grounded actions. Scheduling must not be confused with planning, since the latter regards *which* actions must be performed.

Task
Task is a generic term that indicates which activity the robot should perform. Depending on the level of abstraction, it can refer either to the execution of an high-level specification (*i.e.*, the execution of a symbolic plan), or the continuous activity currently performed by the control. If multiple skills can be executed concurrently, then their run-time composition defines a *task specification* that the robot must perform.

Plan Executive
The plan executive, also known as task executive, is the software entity responsible for executing a symbolic plan, thus grounding and scheduling a set of motion skills by composing a task specification that the robot must perform.

Three Tiered Control Architecture
The term *Three Tiered Control Architecture* (3T-A) is a widely known architecture in Robotics, firstly suggested in [24, 83] and adopted by several

frameworks [147, 148, 3, 125]. The 3T-A model assumes the existence of a third layer between planning and control, called *task executive* or *plan executive*, responsible for coordinating and scheduling the various activities on the robot starting from a given plan. This dissertation focuses mainly on the task executive layer, but it also expands investigations in planning and control directions, showing that this separation is not so strict.

1.3 Methodology

This section briefly resumes the hypotheses and the tools adopted to pursue the objectives of this dissertation.

1.3.1 Skill-based Approach

This work advocates the use of skills to ground a symbolic action. Conceptually, a skill is a reusable strategy that implements a symbolic action in a similar *context*. This concept is known in literature, but its concrete incarnation changes between frameworks and research areas. For instance, *learnable motion skills* are extracted from human demonstrations and encoded as parameterisation over motion primitives [149, 7, 114, 75, 87]; *programmable skills* are defined by *procedural* programming [121, 23, 162, 68] or *constraint-based* programming [151, 150]; recent investigations aim to extract constraint-based task specifications from human demonstration which can be used by other solvers [163].

A skill execution exhibits both *discrete* and *continuous* behaviours, so the skill suits to the role of intermediate primitive between a symbolic action and its implementation. As the scheduling of a symbolic action depends on some logical *(pre-)conditions*, a skill execution depends on a set of constraints that reflects the symbolic order of the plan; however, those dependencies are not expressed in the discrete domain, but they must link to the continuous domain.

A skill model adds formal knowledge into an action description, and multiple skills can implement the same action on different contexts. The *context* plays a fundamental role, since it provides the necessary information that makes the skill deployable and executable. Figure 1.4 shows the knowledge that defines the context: *i)* the robot capabilities, *e.g.*, tools and sensors equipped to the robot; *ii)* the intention of the action; *iii)* the objects that involves the action; *iv)* the environment that constrains the execution of the action. Such a knowledge is involved in any aspect of a robotic application, from the symbolic planning to

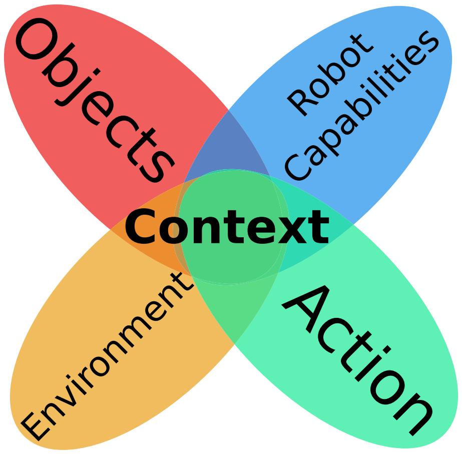


Figure 1.4: The knowledge that composes a context of a skill execution.

the generation of the motions, but with different *views* or *level of abstractions*. Therefore, the complete knowledge may not be fully available at the beginning of the plan execution, but could rather be acquired during the execution itself. Concretely, this research promotes the adoption of different phases to ground an action to an executable skill, as shown in Figure 1.5: a first refinement process grounds a symbolic action to a skill prototypes, and then the skill prototype is fed with additional knowledge. As a result, the same action can be grounded by many skill prototypes, which in turn they boil down to a different skill instance, depending on the knowledge of the context in which the action is performed.

Finally, yet another research hypothesis concerns the *composability* of a skill, both from the modelling and the execution point of view. The former advocates that a skill can be modelled as a composition of other skills, while the latter indicates that skills can be executed concurrently.

1.3.2 Constraint-based Approach

One of the objectives of this dissertation is to employ declarative specifications rather than procedural programming. To this end, a proposed methodology is to consider everything as a *constraint-based* problem. This is motivated by the fact that most of the solvers, in both discrete and continuous domains, already adopt such a methodology:

- in the planning domain, a plan is generated from the knowledge about

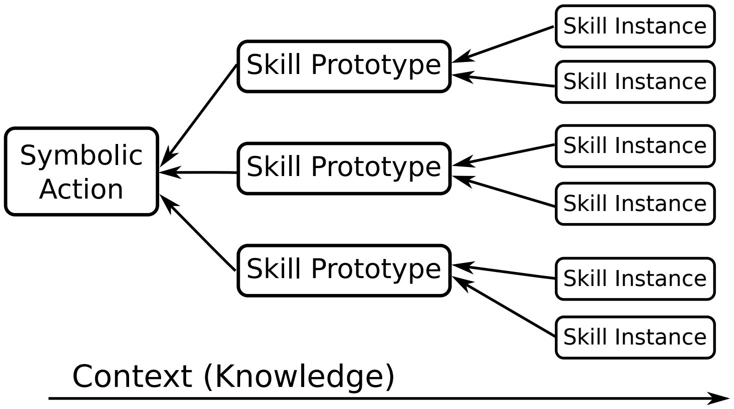


Figure 1.5: Simplified representation of the phases that ground an action. An action can be implemented by multiple skill prototypes, which in turn are grounded as a skill instance. The arrows between action and skills denote a *conform-to* relationship, while an additional arrow indicates the increasing detail of the knowledge required in the refinement process.

possible actions and classes of objects that populate the world (the *"domain"*), and the concrete world instance and goals to achieve (the *"problem"*). Both *domain* and *problem* can be expressed as a set of logical constraints, formulating a *Constraint Satisfaction Problem* (CSP);

- in the control domain, *Constraint Optimisation Problems* (COPs) are used to describe a motion problem, whether the solution is a trajectory to follow or an instantaneous control action to apply to the robot joints; the former regards *motion planning*, the latter concerns *motion control*. This solution aims to minimise a certain *objective function* that reflects the intended behaviour, while respecting some constraints, *e.g.*, about the robot capabilities, such as joints limits.

These similarities lead the constraint-based approach to be a candidate as an "interlingua" between discrete and continuous problems, where logical constraints are translated to motion constraints (and *viceversa*). This hypothesis has been investigated already, bridging the symbolic planning to the *motion planning*; examples are in [18, 47, 74, 156, 89, 88]. However, these approaches still reside in the planning domain, where everything must be known in the planning phase. This leads to some assumptions and compromises: all the knowledge must be available in advance, thus the environment is static; otherwise replanning is necessary, but its computational cost is extremely high in a realistic scenario, and such a cost increases even more with the introduction of motion constraints.

Furthermore, even if dynamic constraints can be taken into account in a generic motion planning problem, none of the above-mentioned prior-work considers force-based tasks. Approximations that allow geometric reasoning over physical properties of an object is possible [106], but also in such cases interaction tasks cannot be detailed. Resuming, motion planning techniques can be adopted to reason about the feasibility of a plan, but not for active control.

Instead, this dissertation focuses on bridging symbolic plans to those COP techniques that perform reactive control, such as the *instantaneous Task Specification using Constraints* (iTaSC) [36], the *Stanford Whole-Body Control* (SWBC) [141], the *Stack of Tasks* (SoT) [96] or the recent *expressiongraph-based Task Specification Language* (eTaSL) [2]. To the best of the author's knowledge, this promising synergy has been only partially explored [120, 121, 12], probably due to the different research interests of the Robotics Community.

1.3.3 Graphs Models for Knowledge Representation

Graphs are known data structures in AI [32, 169], Control and Robotics literature, mostly used to store knowledge and relationships between the connected information; some examples are: graph structures for CSP [39]; world model representation [22]; algorithms for *Simultaneous Localization and Mapping* [64]; *bond-graphs* [6, 115]; semantic description of kinematic chains [142]. This dissertation is not an exception, and it promotes the adoption of *hierarchical hypergraphs* as a primary *structural model*:

- a **hyperedge** is a first-class citizen that represents a n-ary relationship;
- **nodes** are hierarchical, so they can contain a graph. This introduce the generic concept of *level of abstraction*, since each node can be expanded to contain deeper knowledge, which is relevant only for some parts of an application, or unknown *a priori*.

Additional structural constraints can be imposed by the specific knowledge domain that the graph represents; in this dissertation, such a knowledge embeds relationships between executable motion skills. This work advocates the importance of separating the *structural* model from its interpretation (*i.e.*, behaviour and functionalities), which is domain-specific.

1.3.4 Model-Driven Engineering Methodology

Model-Driven Engineering is a software methodology to create and describe architectural solutions by means of domain specific models [80, 19]. Instead of describing the functionalities through *Application Program Interfaces* (APIs), whether as routines or as protocols, in this work the representation choice falls in the *Domain Specific Languages* (DSLs) [55]. In this way, both structural data and attached semantics are decoupled from the underlying implementation of the functionalities, which allow to convey the achieved results in a formal manner. Atkinson *et al.* [8] suggest four modelling layers that represent the different abstractions over the information that the model contains:

M3: the *domain-independent model*, also called *meta-meta-model*, represents the generic and invariant concepts and relations that hold over multiple domains. In MDE, it is also referred as a language or the tool to model other languages. In the context of this work, an example is the mathematical foundation of the graph theory, since graphs are employed to represent structural information (vertex) and the relationships between those (edges);

M2: the *domain-specific model*, or simply *meta-model*, augments a M3 model with domain-specific information and constraints. This work proposes DSLs that reside in this layer, since they are meant to solve a problem in a specific domain. An early example is the structural rules applied to a generic graph to describe the dependencies between executable skills;

M1: the *domain model*, or just *model*, is a specific instance of a M2 model, for example, a specific set of dependencies between skills;

M0: the *instance* of a M1 model that often represents the concrete information of the above layers.

These abstraction layers are related to each other by a *conforms-to* relationship, from which entails a statement of software-reusability: each functionality implemented in one specific M3 model can be applied to any M2 model that *conforms-to* the M3 model, and so on for the other layers. Specific details are discussed for each DSL proposed in this dissertation.

1.3.5 JSON and JSON-Schema

Another decision over a DSL design is whether its implementation is internal (or embedded) to a general purpose language. The initial development of the DSLs

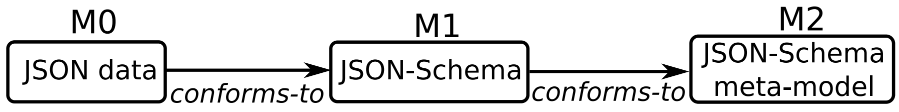


Figure 1.6: The *conforms-to* relationship and the modelling layers in the context of JSON documents.

in this dissertation is supported by the Lua scripting language [70] to obtain executable models. However, this work presents alternative representations based on *JavaScript Object Notation* (JSON) [28] documents. The JSON data format suits the requirements of a format for sharing models, since it is language-independent and both machine and human readable. Furthermore, the structural representation of a JSON document can be described by a specific *schema* model, which resides in the M1 level. In turn, such a model conforms to the so-called JSON-Schema [57], a meta-model (M2) that defines the semantics of the keywords used in a schema, also encoded as a JSON document[2]. This approach enables structural checks over the validity of a JSON document. Figure 1.6 depicts the modeling layers around the *conforms-to* relationships in the JSON context, while Figure 1.7 proposes a concrete usage example. Finally, the recent JSON-LD (Linked Data) [155] allows to link both data and models together, such that new DSLs can be obtained by *composition*. This dissertation does not make use of the JSON-LD, but such a tool further motivates the JSON choice.

1.4 Contributions and Outline

This research proposes formal methods to ground a symbolic action to an executable motion skill, and to compose and to coordinate the concurrent execution of multiple skills. The skill scheduling is driven by logical constraints, such that the behaviour is adapted to the context of the execution. In detail, this dissertation provides a structural way to model both discrete and continuous aspects of a skill, taking into account not only the intended behaviour, but also non-nominal situations. The composition of the skills to be performed defines a constraint-based task specification online, which in turn generates a constraint-optimization problem; the computed solution is the control action to apply to the robot joints. As a whole, this work can be seen as an *integration* effort to combine symbolic plans and executable task specifications. For this reason, this dissertation does not contribute to improve the current state-of-the-art

[2]The specification of the JSON-schema meta-model can be found in `http://json-schema.org/draft-04/schema`.

```
{
 "name" : "joint1",
 "position": 0.21,
 "velocity": 0.04,
 "acceleration": -0.01,
 "timestamp":
     1456389234,
  ...
}
```

```
{
  "id": "http://.../joint_info",
  "$schema": "htttp://json-schema.org
      /draft-04/schema",
  "description": "joint information"
  "type" : "object",
  "properties" : {
    "name" : {"type" : "string"},
    "position" : {
      "type" : "number",
      "minimum" : -0.3,
      "maximum" : 0.3
    }
    ...
  }
}
```

Figure 1.7: A JSON document that reports some data specific to a robot joint (M0, on the left) that conforms to a JSON-Schema model (M1, on the right), which in turn conforms to the JSON-Schema meta-model.

with respect to solvers and algorithms in both planning and control domains. Instead, it provides a novel programming paradigm that bridges the gap on the scheduling of discrete plans in the continuous domain. Moreover, this work addresses a major issue that affect most of the current control frameworks, which is the limitation of composing and configuring critical parts of an application in an early stage, often making decisions that are not optimised to the current context of the execution.

In detail, each chapter contributes to address the several questions that compose the problem statement (Section 1.1) as reported below.

Chapter 2 introduces to the *Skill Dependency Graph* (*SDG*), an executable model that grounds a symbolic plan by means of the dependencies (edges in the graph) of each skill (vertex in the graph) representing a symbolic action. To this end, this chapter proposes a skill model that represents both nominal and non-nominal situations, and that bridges a discrete description to a continuous motion specification (**Q4**). The *SDG* model, interpreted by an executive engine called *Skill Dependency Graph Executive* (*SDG*-E), is built through a minimal set of declarative rules (**Q7**) that expresses the relationships that constrain the execution of each skill; this approach defines a language called *micro Skill Dependency Language* (*uSDL*). Therefore, the skill execution is influenced by logical conditions on the continuous state of both robot and environment: a contribution consists in formally defining such a mapping (**Q5**, **Q6**). At run-time, the satisfaction of the modelled dependencies determines whether the skills are executed concurrently or not, that is, defining an execution order (scheduling) online (**Q2**).

Chapter 3 proposes a constraint-based task specification using geometric expressions. A geometric expression is a relationship between geometric entities that are part of a model of the manipulated object, such as points, lines and planes. This object-centric DSL extends existing task specifications, which are often coupled to a specific control strategy. This DSL contributes to decouple the task specification from the underlying numerical solver. This chapter partially contributes to describe the grounding of an action in the continuous domain (**Q3**, **Q5**).

Chapter 4 contributes to integrate the previous findings, aiming to generalise the description of a skill model. The result is a *skill prototype*, a skill having each behaviour described by a constraint-based motion specification, which in turn is defined on the basis of generic geometric items (**Q3** and **Q4**). A *skill prototype* is not executable *as is*, but it requires additional *context-dependent* knowledge. Another contribution is the definition of the *context-dependent* information (**Q1**) that, once available, refines the skill prototype to an executable skill instance. This chapter is accompanied by a set of examples showing how to define complex skill prototypes by *composition* of existing ones.

Chapter 5 addresses to the question **Q8**, proposing a *"Just-in-Time"* (JIT) strategy to compose a motion specification at run-time on the basis of the *SDG* model and a *Skill Life Cycle*. This approach aims to postpone all the decisions until it is needed, so that the action is grounded to a skill that encodes the latest, context-dependent information available at run-time. In addition, this chapter shows how to delegate decisions often taken during the planning phase to a local *SDG*-E. Therefore, this chapter partially contributes to solve the questions **Q1**, **Q2** and **Q3**.

Chapter 6 advocates the *hierarchical hypergraph* as a structural model to describe context-dependent knowledge. The objectives are: *i)* to provide a formal language to describe hierarchical hypergraphs, called *NPC4*, *ii)* to separate the structural model of a graph from its behaviour (or functionality), in this way *iii)* other DSLs can be realised by adding domain-dependent constraints on the structure, then "activated" by domain-dependent functionalities. This work contributes to standardise graph-based models, by separating common from domain-dependent assumptions, thus promoting model sharing and reusability of common functionalities. This modelling effort applies to a broader scope than the one discussed in this dissertation, and it provides further insights on the *SDG* approach, since the *SDG* structural model conforms to the *NPC4* language. That is, this chapter is an answer to **Q9**.

Appendix A reports a set of models and meta-models in JSON and JSON-Schema format that are discussed along the text.

Suggestions to the reader: a first set of dependencies concerns the reading order of the chapters. As they are, the chapters are ordered to facilitate the understanding of the solution proposed. However, this order is not strictly necessary, since Chapter 2 and Chapter 3 are self-contained and their order can be reversed: this is yet another partial-order plan. The author crafted this text taking into account the heterogeneous background of the treated concepts, and it is author's hope that the reader will enjoy this dissertation.

Chapter 2

Skill Dependency Graph

This chapter introduces the *Skill Dependency Graph* (*SDG*), a first modelling step in a broader methodology to coordinate, configure and compose robotic behaviours. A *SDG* is a model that represents a concrete instance of a plan, where the *skills* are executable primitives that exhibit both discrete and continuous behaviours. The skill execution is driven by a set of *dependencies* that must be satisfied during or prior to execution; their online evaluation is based on monitored conditions that capture the dynamic changes of the world. In this way, the *scheduling* of the skills is determined by the *context* in which the robot operates, ordering their activation accordingly. The *SDG* model suits to represent situations, that is the execution order of the skills (*i.e.*, scheduling), described by a set of declarative rules, as an alternative to procedural recipes.

To this end, *logical conditions* and skills play a relevant role, so a first formalisation is required (Section 2.1 and Section 2.2). On the basis of the latter, a formalisation of the *SDG* model, its primitives and its relationships is presented in Section 2.3. The online execution of a *SDG* model is coordinated by the *Skill Dependency Graph Executive* (*SDG*-E, Section 2.4), a software entity that resides in the middle layer of a *"Three-Tiered"* control architecture [24, 83]. To compose a *SDG* model and to express the skills scheduling order by declarative rules, a language called *micro Skill Dependency Language* (*uSDL*) is introduced in Section 2.5. The features of the proposed approach are demonstrated by the solution of a concrete *Open a Drawer* use-case (Section 2.6). Finally a review of related works (Section 2.7) reports about alternative mathematical formalisms and concrete plan (or task) *executives*.

2.1 The Execution Monitoring Role

Execution monitoring is a key feature of any automated system; it allows to detect and classify the quality of the performed behaviour. That may differ from the *nominal* case, and the causes of such differences may vary: a hardware failure, uncertainties, unreliable resources, environment changes, unforeseen situations and so on. Some of those issues have been widely investigated in the past by the control community, referring to the problem of *fault detection and isolation* (FDI). These solutions have been successfully applied to industrial and aerospace domains, but not in the robotic context. A survey of the existing approaches in robotics is presented in [119], which identifies three categories: *analytical*, *data-driven* and *knowledge-based*. This separation is mostly due to an heterogeneous background of the community involved in the robotics research. Anyway, they share a common definition of execution monitoring, namely the online detection of anomalies in the behaviour of a system.

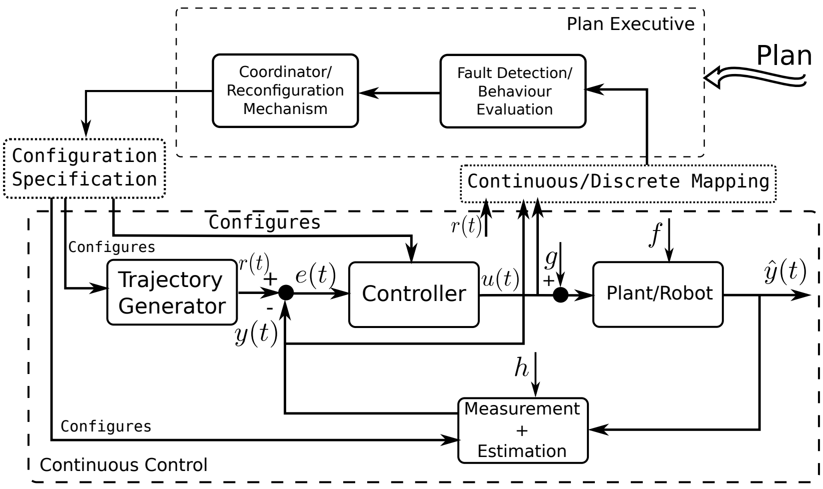


Figure 2.1: Architectural separation between a continuous control strategy (bottom) and a discrete plan executive (top). A generic closed loop strategy is shown in form of a control diagram: r and u are reference signal and control action, respectively; g, f and h represent the disturbances in the continuous domain. The plan executive generates a specification that configures the control algorithms. Such a configuration ensures a correct execution of a plan, considering the logical information from the underlying control system, which must be mapped from the continuous domain.

In the context of this work, a contribution aims to provide a link between a *continuous* monitor and the symbolic representation from a plan executive perspective[1]. Considering a generic system in Figure 2.1, a *plan executive* coordinates and configures the control system, in order to perform the desired behaviour of a given plan. A configuration mechanism interacts with the controller by changing its settings, selecting a different behaviour, or even replacing the control algorithm strategy. Such decisions are made upon a previous evaluation of the behaviour quality performed with respect to a nominal plan, which is often specified as a plain sequence of tasks. Therefore, execution monitoring has a relevant role, since it is the primary source of information to evaluate the success or the failure of a motion. The more the monitor information is rich and precise, the more the controller configuration is adapted to unforeseen situations.

However, no special care is often taken on the role of the execution monitoring, and this is denoted by the lack of appropriate primitives in the current *task specifications*. Task specifications are those that describe a motion, often an object manipulation, independently from the agent that will perform it. Most of the current state-of-the-art in Robotics only focuses on the *nominal* scenario, defining only those conditions that determine a success. A typical example is the usage of *guarded-motions* in the Task Frame Formalism (TFF) [30, 82], in the Listing 2.1; the condition, which is expressed in the last line of the specification, determines the *nominal* success of the motion. Note that the TFF formalism does not provide a linguistic primitive to denote *non-nominal* terminations[2], such as a timeout, a failure on the performed behaviour and so on.

```
move compliantly {
  with task frame directions
  xt: velocity 0 mm/sec
  yt: velocity 0 mm/sec
  zt: velocity v_des mm/sec
  axt: velocity 0 rad/sec
  ayt: velocity 0 rad/sec
  azt: velocity 0 rad/sec
} until zt force <- f_max N
```

Listing 2.1: Example of a guarded-motion task definition from [30].

A step-forward in this direction is provided by the *expressiongraph-based Task Specification Language* (eTaSL) [2]. eTaSL offers the capability to define multiple event-based monitors from an arbitrary numerical expression; an example is shown in Listing 2.2.

[1]In this work, the terms *"plan executive"* and *"task executive"* are used interchangeably.

[2]This is not the only hidden assumption of TFF; another assumption is against *robot capabilities*, e.g., a robot should be able to move in 6 DOF, capable of sensing forces and so on.

```
Monitor{
  context = ctx,
  name    = "goal_reached",
  expr    = norm(origin(arm)-origin(goal)),
  lower   = 1E-4,
  actionname = "event",
  argument    = "e_goal_reached"
}
```

Listing 2.2: Code snippet of a monitor specification in eTaSL [2].

This mechanism provides higher expressivity on defining the monitored quantity, but the handling of the generated events is left to the user. In fact, eTaSL promotes a FSM-based coordinator that reacts to those events, linking a continuous behaviour to a discrete one. However, compiling a FSM is a *procedural* form of programming; the FSM is usually hand-written, thus it is statically defined. Therefore, it is not trivial to model all possible non-nominal situations over a complex application.

The importance of an execution monitoring system has been underlined in the architectural *Composition Pattern* introduced in [167, 165]. In the composition pattern, a monitor is a *functional entity*, responsible for verifying some conditions and raise events accordingly. A concrete DSL implementation that complies to the *Composition Pattern* is introduced in [168]. However, this DSL does not provide any formal facility to define a criterion to detect a failure of the task under execution, and it falls on the same already mentioned limitations of the eTaSL framework. The following sections aim to address such limitations.

2.1.1 Monitors and Conditions in the Continuous and Discrete Domain

This section proposes a formal tool to bridge the monitoring of a continuous quantity to a discrete description. In control theory, a **signal** is any quantity exhibiting variation in time (or space) that conveys specific information about a behavior or a status of a physical system. In a robotic context, examples are: (i) a position, (*e.g.*, joint position, an object position) (ii) a velocity, (*e.g.*, joint velocity, an object velocity) (iii) a force, (*e.g.*, joint torque, contact force against a surface). In Artificial Intelligence (AI), the **fluent** has the same meaning of signal, that is anything whose value is subject to change over time. Actually, in AI the term fluent is often limited to *propositional fluents*, that is a **logical condition** whose truth value changes over the time. One example is the proposition *"the drawer is open"*: such a condition may hold or not, and its evaluation can be *grounded* in many forms, e.g., evaluating the relative position of both drawer and furniture. However, the *threshold* that determines if the drawer is open may vary, depending on the *domain of the application*; e.g., a small opening suffices to place small items in the drawer, otherwise the drawer must be fully open for placing large objects. Furthermore, a spatial relationship is not the only way to ground the truth value of such a proposition. In fact, an agent may verify this condition with a concrete action, such as moving the drawer along the closing direction, and checking the forces of such a physical interaction. In short, the mechanism of monitoring a quantity, often called *symbolic anchoring* or *lifting*, is rather complex, since involves *knowledge*, *context* of the application, *perception* and related execution of *motions*. This work does not focus on the overall anchoring problem, but on the monitoring necessary to validate the expected effect of a motion, bridging continuous and discrete domains.

Resuming, monitoring a quantity means to evaluate and to assign a semantic information to its value over the time. For this purpose, a *monitor function* is introduced as follows.

Monitor Function: let $y(t)$ be a signal representing the quantity to monitor, then a monitor function m is defined as

$$m(y(t), \zeta(\cdot)) = \begin{cases} \textbf{True} & \text{if } y(t) \leq \zeta(\cdot) \\ \textbf{False} & \text{otherwise,} \end{cases} \tag{2.1}$$

where $\zeta(\cdot)$ is a *threshold function* associated with a relational operator $(<, \leq, =, \geq, >)$ that converts the *continuous* value of y to a *logical* truth value.

In short, the monitor function converts a signal[3] to a fluent having the semantic meaning of the *condition* that it represents. Furthermore, multiple monitor functions can be applied on the same signal $y(t)$, representing different logical propositions. To simplify the terminology, we drop the term *fluent* in favor of the generic term *condition*. This is justified by the fact that all the logical propositions are time-varying in the context of this work.

Logical Condition c is a symbolic variable that represents the truth value of a logical proposition. A condition is linked to a monitor function that grounds the evaluation of the truth value, formally $c \Leftarrow m$.

The difference between a monitor function and a condition is the domain in which they are defined. A condition c lies in the symbolic domain, while the

[3]Of course, it is assumed that continuous time signals are actually elaborated in a digital domain, as signals sampled at a proper frequency, without need to introduce a change of notation (*e.g.*, from t to t_k).

associated monitor function m is the concrete anchoring mechanism in the continuous domain. Of course, the evaluation of c is possible only if it is mapped to a monitor function m. However, the condition c abstracts from its grounding, and the mapping with a monitor function can be postponed until an evaluation is required. To re-call an evaluation of c in the symbolic domain, the predicate `holdsAt` is borrowed from *Event Calculus* and *Situation Calculus* [145] as follows:

***holdsAt(c,t)* predicate:** a condition c holds at time t. Formally,

$$\texttt{holdsAt}(c,t) \leftarrow m(y(t), \zeta(\cdot)) \tag{2.2}$$

The `holdsAt` predicate implies a duration over the time of its truth value, thus it is not *instantaneous* (the instantaneous concept is provided by the *event* primitive, introduced below). Therefore, if $\texttt{holdsAt}(c,t)$ is true, then it must exists a time interval for which the same predicate holds; formally:

$$(\texttt{holdsAt}(c,t) \leftarrow True) \rightarrow \exists\, t_1, t_2 : \forall t_i \in [t_1, t_2[\; \texttt{holdsAt}(c,t_i) \leftarrow True \tag{2.3}$$

The latter motivates the introduction of a τ parameter that represents the minimum time interval that captures the truth value of a condition[4].

Furthermore, new conditions can be defined starting from existing ones through a logical composition. In short, a condition can be defined as *boolean expression* that composes a predicate. For example, a condition c_3 can be defined by the following expression:

$$\texttt{holdsAt}(c_3,t) \leftarrow (\texttt{holdsAt}(c_1,t) \wedge \texttt{holdsAt}(c_2,t)).$$

Another relevant concept is the **event**, which is strongly coupled to a condition evaluation.

Event e: an *instantaneous* notification of a change over time of the truth value of a condition c.

Formally, an event is yet another fluent that holds only at one very specific point in time. Figure 2.2 shows a sequence of events defined over a condition c. The function $\texttt{events}(c)$ returns an ordered set $\mathbb{E}$ of events defined on c, while the function $\texttt{event}(c,j)$ returns a specific event $e_{c,j}$. The truth value of an event is evaluated by the predicate `occurs`, that is:

occurs$(e_{c,j}, t)$ **predicate:** the event $e_{c,j} := \texttt{event}(c,j)$ occurs at time t.

[4]Such a parameter does not necessarily correspond to the sampling time of the signal, which is the higher time resolution for the execution monitoring.

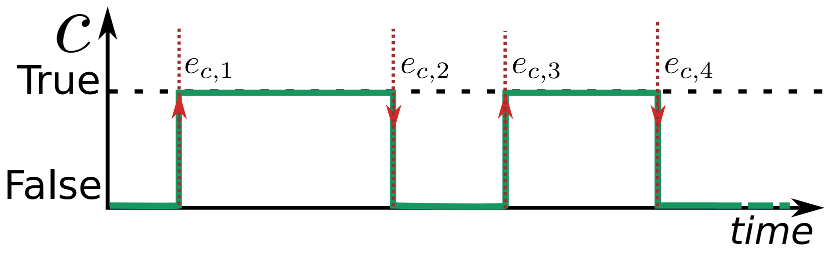


Figure 2.2: A sequence of events $\mathbb{E} = \mathit{events}(c) = \{e_{c,1}, e_{c,2}, e_{c,3}, \dots\}$ over a condition c. $e_{c,1}$ and $e_{c,3}$ are *rising* events; $e_{c,2}$ and $e_{c,4}$ are falling events.

The above-mentioned definitions and predicates are propositional *axioms*, which are arbitrary and refutable by nature. Alternative definitions exist in literature, all with different implications. For instance, in *Event Calculus* conditions (fluents) and events are fully decoupled concepts. However, the motivation behind those definitions is given by a *bottom-up* approach: a condition is defined through a monitor function that bridges discrete and continuous domains. As it will be discussed in Section 2.7, the level of expressivity does not change with respect to other approaches, but the adopted axioms directly link to monitored quantities.

Given the previous definitions, then the following axiom implication is well-formed:

$$\begin{aligned}\texttt{occurs}(e_{c,j}, t) \leftrightarrow &(\neg\texttt{holdsAt}(c, t^-) \wedge \texttt{holdsAt}(c, t^+))\\ &\vee (\neg\texttt{holdsAt}(c, t^+) \wedge \texttt{holdsAt}(c, t^-))\end{aligned}$$

where t^- and t^+ defines a neighbourhood on t.

As shown in Figure 2.2, a distinction between *rising* and *falling* events exists, that is

$$\texttt{eventRising}(e_{c,j}, t) \leftrightarrow (\texttt{occurs}(e_{c,j}, t) \wedge (\texttt{holdsAt}(c, t^+)))$$

$$\texttt{eventFalling}(e_{c,j}, t) \leftrightarrow (\texttt{occurs}(e_{c,j}, t) \wedge (\neg\texttt{holdsAt}(c, t^+))).$$

Furthermore, events defined over the same condition are unique in time, formally

$$\forall C, t : \texttt{occurs}(e_{c,j}, t) \rightarrow \forall k,\ k \neq j,\ (\neg\texttt{occurs}(e_{c,k}, t))$$

Another interesting predicate is the `happened`, which evaluates if the event has occurred in the past.

happened(e, t) **predicate**: given an event $e_{c,j}$ defined over a condition c, and a time $T \in \mathbb{R}^+$, then

$$\texttt{happened}(e_{c,j}, T) \leftarrow \exists t \in \mathbb{R}^+,\, t \leq T : \texttt{occurs}(e_{c,j}, t)$$

Conditions, events and related predicates are sufficient primitives to construct a complex monitor expression. On the same quantities can be applied different monitor functions, which are linked to logical conditions in the discrete domain. The meaning behind the truth value of a condition is *context-dependent* and interpreted by the execution monitoring system. As an example, Table 2.1 shows multiple monitor functions that aim to evaluate a success condition c_{ok} and a failure condition named c_{fail}. The success condition holds if the monitor quantity $y(t)$ lies in a range of values within a given timeout T_{out}. In addition, the dynamic evolution of $y(t)$ is taken into account, that is the condition holds if $y(t)$ does not vary. However, if the timeout occurs, $y(t)$ lying in the given range is acceptable. On the other hand, c_{fail} holds if the latter has not been obtained. Table 2.2 depicts a possible evolution of the monitored signals using the monitor functions in Table 2.1. The above-described *semantics* represent a common pattern for monitoring a quantity that exhibits a dynamic behaviour. Other *context-dependent* interpretations are, of course, feasible.

1	$m_{up}(y(t), \zeta_{up}(t)) = \begin{cases} \textbf{True} & \text{if } y(t) \leq \zeta_{up}(t) \\ \textbf{False} & \text{otherwise} \end{cases}$ $m_{low}(y(t), \zeta_{low}(t)) = \begin{cases} \textbf{True} & \text{if } y(t) \geq \zeta_{low}(t) \\ \textbf{False} & \text{otherwise} \end{cases}$ $c_{up} \leftarrow m_{up}, \quad c_{down} \leftarrow m_{down}$
2	$m_{\dot{y}}(\dot{y}(t), \zeta_{\dot{y}}(t)) = \begin{cases} \textbf{True} & \text{if } t \leq \zeta_{\dot{y}}(t) \\ \textbf{False} & \text{otherwise} \end{cases}$ $c_D \leftarrow m_{\dot{y}}$
3	$m_{T_{out}}(t, T_{out}) = \begin{cases} \textbf{True} & \text{if } t \geq T_{out} \\ \textbf{False} & \text{otherwise} \end{cases}$ $c_{T_{out}} \leftarrow m_{T_{out}}$
4	$\texttt{holdsAt}(c_{in}, t) \leftarrow$ $\texttt{holdsAt}(c_{down}, t)) \wedge (\texttt{holdsAt}(c_{up}, t))$ $\texttt{holdsAt}(c_{ok}, t) \leftarrow$ $\texttt{holdsAt}(c_{in}, t) \wedge$ $(\texttt{holdsAt}(c_{t_{out}}, t) \vee \texttt{holdsAt}(c_D, t))$ $\texttt{holdsAt}(c_{fail}, t) \leftarrow$ $\texttt{holdsAt}(c_{t_{out}}, t) \wedge (\neg\texttt{holdsAt}(c_{in}, t))$

Table 2.1: Monitor functions applied to the signals shown in Table 2.2, as an example of a composite monitor of the predicate *"$y(t)$ signal is in range $[\zeta_{down}, \zeta_{up}]$ before a timeout T_{out}"*. $\dot{y}(t)$ derivative of the signal is considered to check if the evolution of y is steady. The predicate is evaluated from the condition c_{ok}, defined as boolean expression over the conditions $c_D, c_{T_{out}}, c_{up}$ and c_{down}. As a result, c_{ok} holds if y lies in the desired range before T_{out}, but c_D condition also avoids to trigger an event $e_{c_{ok},1}$ on the range border, if y is still changing. The above-described mechanism is a useful pattern to properly determine the success (or failure) of a grounded action. All the threshold functions are expressed as a constant value.

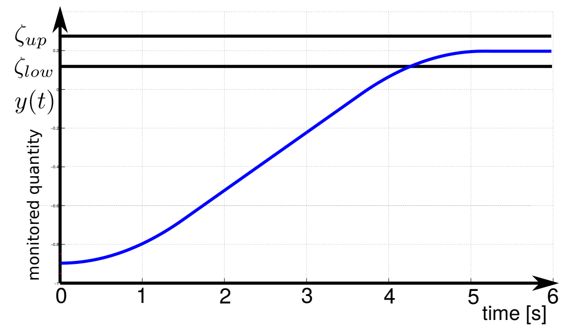
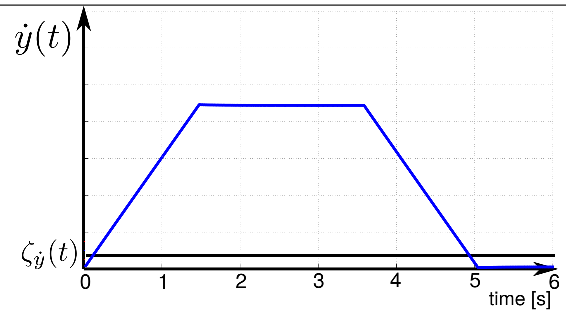
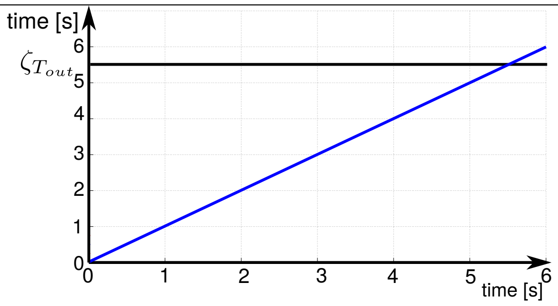
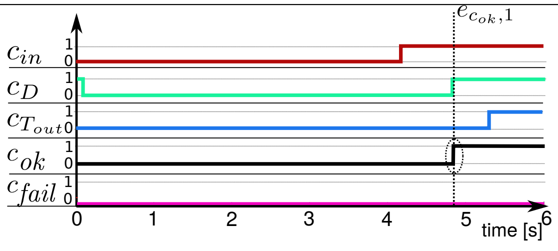
1	
2	
3	
4	

Table 2.2: Example of evolution of a set of monitored signals and conditions; the monitor functions applied to the monitored signals are illustrated in Table 2.1. In this example, the condition c_{ok} holds, since the monitored signal $y(t)$ is in range $[\zeta_{down}, \zeta_{up}]$ and steady within the timeout T_{out}.

2.2 The Skill Model

This section proposes the concept of the **skill** as a formal model to cover both discrete and continuous representations of a grounded action. The purpose of an action is to perform a desired world-transformation, sometimes called intended **effect** or **post-condition**. In literature, the *execution* of a symbolic action can lead to one of the following status: *i)* successfully executed, *i.e.*, the intended effect is achieved; *ii)* the execution fails, *i.e.*, the intended effect is not achieved; *iii)* the action is planned but not yet executed; *iv)* the action is under execution but its result is not known yet. This discrete abstraction is a consequence of a *functional* approach widely used in the symbolic domain: the execution of the action is delegated to an underlying controller, and a plan executive decides the next action to execute upon the current outcome. This strategy does not permit to reason about the execution of the action itself, since there is no information about *"what is happening during execution"*: the skill model addresses this limitation. The execution of the proposed skill model aims to achieve the same intended effect of the action that the skill grounds. Moreover, a skill (execution) monitoring captures the dynamic behaviour of the execution, which is systematically mapped in the symbolic domain. This allows to further extend the reasoning about the motion execution itself locally, at the plan executive level.

2.2.1 A Motivational Example

Once an action is activated, the intended effect is not obtained immediately, but in accordance with the dynamic that involves all the actors in the execution, whether they are controlled (*i.e.*, the robot) or not (*i.e.*, the environment). For a better understanding, let us consider a concrete *pouring water* case of study. A robot serves a glass of water, filling 25 cl of water from a jug. For such a task, the main action consists in pouring the water, and that can be achieved by tilting the jug. An ideal flow is obtained by a tilt angle of 1.22 rad, but an inclination of 0.87 rad is already enough, due to the water contained in the jug. Such information is strongly **context-dependent**: another jug or a different initial quantity of water would change these values. Before pouring, it is necessary that the jug's spout is on the top of the glass, otherwise the water will be spilled out. This describes a logical constraint, a necessary *pre-condition* that must hold prior to the execution of the action. To this end, another action may be executed to fulfill such a condition, obtaining a plain sequence of action executions.

The execution of a nominal plan is illustrated in Figure 2.3: the condition of

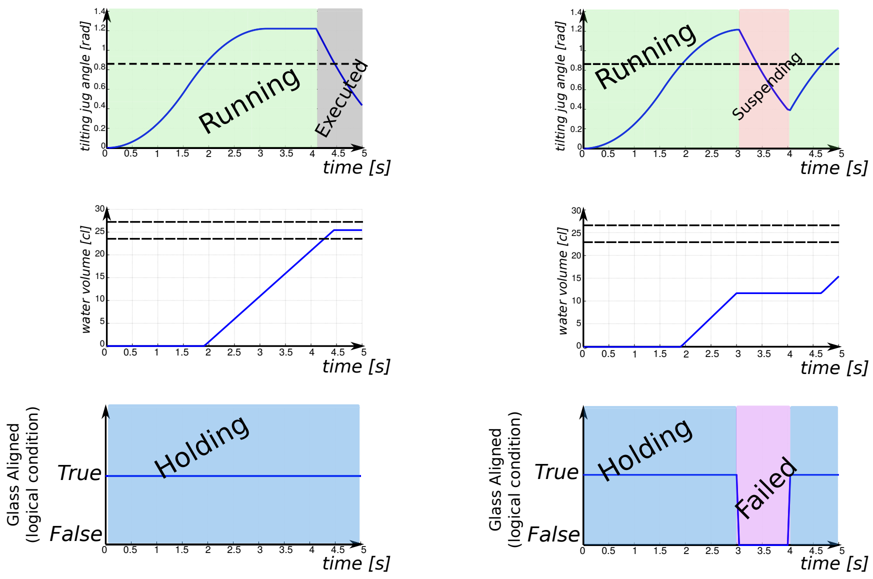


Figure 2.3: Nominal (left) and non-nominal (right) execution of a pouring plan composed of two skills. The nominal execution motivates the status of *running*, *holding* and *executed*. The non-nominal execution, due to a not satisfied logical constraint, motivates the existence of a *suspending* behaviour.

the glass aligned with the jug holds, thus the robot tilts the jug. The skill that refines the tilting action implements a continuous behaviour, illustrated by the controlled tilting angle. The outcome, measured by monitoring the water in the glass, is not instantaneous, but it changes in accordance with the dynamic of the system (jug, glass, water flow, and so on); a monitor function is defined for such a purpose. Once the intended effect (glass filled) is achieved, the execution of the skill terminates, bringing the jug to its initial inclination. This execution exhibits three different discrete status of a skill, each one representing a separate behaviour: i) running status, that is the behaviour that actively tilts the jug, until the intended effect is achieved; ii) executed status, that is the behaviour related to the closure of a skill; iii) holding status, that does not concern the tilting action but a second action that maintains the glass and the jug aligned (if necessary).

However, the nominal description reported above is not sufficient to provide a robust skill model. For instance, no reaction is modelled in case of the glass is moved while pouring the water on it. A human would react spontaneously to such a situation, by tilting back the jug. This is possible because human beings reason on a *open world* context, inferring implicit information given from an assigned task. Instead, automated plans are based on a *closed world* assumption, especially in the executive phase: if an unexpected event is not modelled, the robot may even do not recognise the occurred failure. An example of non-nominal behaviour is reported in Figure 2.3: the alignment condition does not hold, thus the tilting skill switches to a *suspending* status, bringing the jug in a different angle position, enough to prevent to continue to pour (nominal behaviour).

2.2.2 Design Drivers

The previous example illustrates the following aspects that a skill model must consider:

- **hybrid behaviour:** a skill execution exhibits a finite number of continuous behaviours, whether they are *nominal* or not; those can be represented symbolically, that is, a skill manifests both discrete and continuous behaviours, and it can be modelled as a **hybrid system**;
- **composability:** a skill model can be composed of other skills, reflecting the inherent composability of a symbolic action; *e.g.*, a “pouring action” implies the execution multiples sub-actions (*i.e.*, a sub-plan), such as “grasp a jug”, “place the jug’s spout on the top of the glass”, “tilt the jug” and so on; the skill that grounds such an action is implemented by composition of other skills;
- **scheduling:** like actions, it exists an order on the skill execution, mostly driven by condition-based dependencies;
- **context:** an executable skill model adds those context-dependent information necessary for its execution.

2.2.3 Skill Behaviours

A main contribution of this modelling effort is to introduce a minimal but complete set of behaviours that fully describe a skill model. Motivated by the example in Section 2.2.1, the proposed set of behaviours follows:

- *Nominal behaviour*, namely a behaviour that aims to achieve the intended effect of the skill (*e.g.*, tilting the jug);

- *Non-nominal behaviour*, that is a behaviour that replaces the *nominal behavior* in case of non-nominal (but modelled) situations (*e.g.*, to prevent to pour more water if the glass is not aligned with the jug's spout);
- *Holding behaviour*, namely a behaviour that preserves the achievement of an intended effect previously realised;
- *Failed behaviour*, that is a behaviour that expresses what to do in case that the nominal behaviour does not succeed.

The execution of a skill model concerns the selection of the behaviour to apply at *run-time*. An execution engine is responsible for such a decision, which involves a set of declarative rules based on logical conditions continuously monitored. Therefore, the execution engine sets a `status` of a *skill instance*[5] that represents the deployed behaviour; an enumerative list of `status` follows:

- `Inactive`: this is the initial status of a skill instance, not yet in the execution stack; no behaviour is associated to this status;
- `Running`: a modelled *nominal behaviour* is deployed;
- `Suspending`: a *non-nominal behaviour* is applied;
- `Holding`: a *holding behaviour* is deployed;
- `Executed`: is a terminal status that indicates the successful execution of the skill instance; the intended effect is achieved and the skill instance can be dismissed;
- `Failed`: is a terminal status that represents a modelled failure of the skill execution, whether the previous behaviour was nominal or not; in this status, a *failed behaviour* is applied.

As a convention, s indicates a skill instance, while s.`<status>` denotes the run-time property of s, *e.g.*, s.`running`. In the same vein, the fluent `is<status>`(s,t) is introduced to formalise the declarative specifications that rule the skill execution. For instance, `isRunning`(s,t) is a predicate that holds if s is in `running` status at the given time t. During execution, a skill instance s can assume one and only one status at any given time, therefore inferencing as follows is correct:

$$\begin{aligned}\texttt{isRunning}(s,t) \leftrightarrow & (\neg\texttt{isInactive}(s,t)) \wedge (\neg\texttt{isHolding}(s,t)) \\ & \wedge (\neg\texttt{isExecuted}(s,t)) \wedge (\neg\texttt{isSuspending}(s,t)). \qquad (2.4)\end{aligned}$$

[5]A skill instance (M0) is a concrete instance of a skill model (M1).

2.2.4 Hierarchy in the Skill Model

The proposed skill model is *hierarchical*. The hierarchy adds extra semantics to compose several skills as a whole, and it allows to further expand a skill model to a deeper level of detail. The hierarchy relationship imposes a *strict tree structure*: a skill can contain multiple children, but one child has only one parent. To express a hierarchy relationship, the following notation is adopted:

- `contains`(s_1, s_2) denotes that the skill s_1 contains s_2;
- `isParent`(s_1, s_2) is a predicate that holds if a hierarchical relationship exists between the two skills, formally `isParent`$(s_1, s_2) \leftrightarrow$ `contains`(s_1, s_2);
- `parentOf`(s_2) is function that returns s_1 parent node of s_2;
- `childrenOf`(s_1) is function that returns a set of children nodes of s_1.

2.2.5 Logical Conditions

Section 2.1 illustrates the relevant role of the execution monitoring, providing the tools to formalise the mapping between continuous and discrete domain. However, the nature of a condition may vary, *e.g.*, the monitoring over the achievement of a intended effect, the monitoring over the position of an object, or monitoring if a button has been pressed. In the skill model, logical conditions are first-class citizens, and they are classified in accordance with the existence of a *causality* relationship between the execution of a skill and the truth value of the condition. The definitions that follow are part of the skill model, and they are fundamental to constrain logical dependencies between skills, as shown in Section 2.3.

Internal Condition: let c and s be a logical condition and a skill instance, respectively. c is called *internal* with respect to s if there exists a causality relationship between the execution of the skill s (cause) and the truth value of c (effect).

Examples of internal conditions are those that are evaluated by monitoring function(s) defined over a signal(s) controlled by s; *e.g.*, a threshold over a controlled position of an object, a threshold over a controlled velocity of a robotic joint and so on. Other internal conditions are based on a monitored quantity not directly controlled by s but intended as behaviour outcome: an example is a skill that implements a hybrid force/position control. Time-based conditions defined over the execution of s also fall in this definition. For instance, a condition whose the truth value holds after a certain time elapsed in s.`running`.

As a complementary definition, an *external* condition is a condition which is not internal to s.

External Condition: let c and s be a condition and skill, c is said *external* with respect to s if c there is no causality relationship between the execution of s and the truth value of c.

As a remark, this property is always referred to a node s, and it is mutually exclusive, formally $\texttt{isExternal}(s, c) \leftrightarrow \neg\ \texttt{isInternal}(s, c)$.

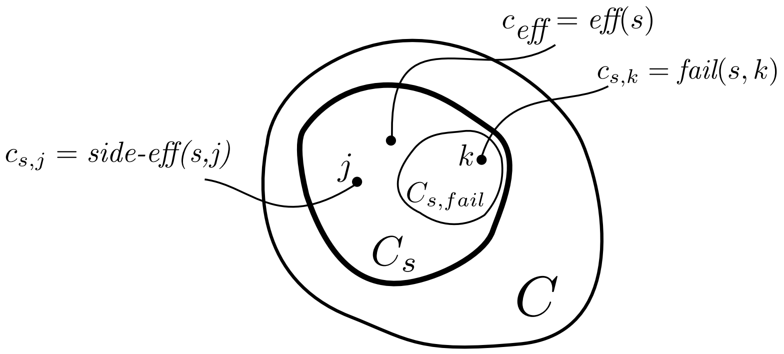


Figure 2.4: Set diagram of the conditions defined over a skill s. C is condition set; $C_s \subseteq C$ represents the *internal conditions* related to s. Within C_s, one and only one condition $c_{\texttt{eff}}$ is *intended effect* of s. Multiple side-effects $c_{s,j}$ exist ($c_{s,j} =$ *side-eff(s,j)*$, \forall c_{s,j} \in C_s \setminus c_{eff}$), among which $C_{s,fail}$ is set of failure conditions.

Given a skill instance s, Figure 2.4 shows the set of internal conditions C_s, which is further classified in:

- **intended effect** (c_{eff}) holds if the goal of the *nominal* behaviour of the skill is achieved. This condition is *unique* for the given skill instance s. However, expressivity is not affected, since such a condition can be expressed as a boolean expression. Formally, the function *eff*(s) refers to the modelled intention of the skill s;
- **side effects** ($c_{s,j}$) are those internal conditions that are not intended *effect* of s. Side effects are not uniquely defined in s; multiple outcomes can be expressed and realised over the execution of s. The notation $c_{s,j} =$ *side-eff*(s, j) indicates the j-th side-effect;
- **execution *failure*** set ($C_{s,fail}$) is a subset of side-effects that reports a skill execution failure. Failure conditions are not unique in s, and the k-th failure condition is indicated with the notation $c_{s,k} = fail(s, k)$. Note

that these failures are *foreseen* failures for which the skill is ready to deal with; *"real"* failures are those for which the skill has no solution modelled.

2.3 Skill Dependency Graph Model

This section presents the *Skill Dependency Graph* (*SDG*), a model that grounds a symbolic plan. A plan is a total or partial-order list of symbolic actions, which are scheduled sequentially or concurrently, depending on the logical constraints that bind their activation. The planning literature provides a rich set of languages to express constraints on the action execution; a selection of these is discussed in Section 2.7. Likewise, skills are connected to each other, executed as a plain sequence or concurrently, depending on some constraints evaluated online. Therefore, grounding a symbolic plan means to ground the planned actions one by one, but also to represent the dependencies that rule the scheduling of the skills. The *Skill Dependency Graph* (*SDG*) is a graph-based model dedicated to represents this information.

Formally, the *SDG* is a *hierarchical, directed acyclic hypergraph* that stores relationships between *skills* through *conditions*. Nodes represent skills, while hyperedges represent conditional dependency relationships between skills, that is, a logical condition that can be evaluated at run-time. The violation of a dependency may prevent a skill execution, or even modify the original behaviour of a skill to a non nominal configuration. The execution of one skill may cause a condition to hold (or not), influencing the execution of other skills. This explains the directed nature of the *SDG*, that indicates if a skill depends on the relationship, or if its outcome may influence the execution of another.

2.3.1 Skill Dependencies

This section formalises the dependencies that drive the execution of a skill instance s. Often defined over *external* conditions of s, these constraints bind the activation of a skill, but also which behaviour of the skill model is deployed during execution. This work proposes two dependency types: *activation dependency* and *invariant dependency*. This separation is due to the different semantic associated to the dependencies, which live in separate contexts. The *invariant dependency* lives *in* the scope of the skill execution, while the *activation dependency* lives *before* the scope of the skill execution. Details on the semantic and related logical proposition axioms are explained as follows.

Activation dependency

The activation dependency is a relationship between a skill s and a user-defined condition c_{start} that indicates *when* the skill can be executed. The condition c_{start} is also called *pre-condition* of s, since it represents the set of requirements that must be satisfied prior to s execution. For this purpose, the predicate $\texttt{isReadyToBeActivated}(s, c_{start}, t)$ holds if the skill s can be executed at time t:

$$\begin{aligned}\texttt{isReadyToBeActivated}(s, c_{start}, t) \leftarrow\ & \underbrace{\texttt{isInactive}(s, t)}_{scope} \wedge \\ & \underbrace{\texttt{isRunning}(\texttt{parentOf}(s), t)}_{hierarchy\ constraint} \wedge \\ & \underbrace{\texttt{holdsAt}(c_{start}, t)}_{requirement}, \end{aligned} \tag{2.5}$$

where:

- the *scope* term denotes that `isReadyToBeActivated` can hold only before the execution of the skill;
- the *hierarchy constraint* indicates that the skill can be activated only if its parent (if any) is under execution and its nominal behaviour is deployed (*i.e.*, `running` status);
- the *requirement* term denotes that the *pre-condition* c_{start} must hold.

As a convention, outside its scope the predicate `isReadyToBeActivated` does not hold. Note that the `isReadyToBeActivated` indicates only if the dependencies on the activation are satisfied; it is the skill executive engine that evaluates this predicate online and actually *activates* the execution of the skill instance (see Section 2.4). For the sake of clarity, Figure 2.5 illustrates the role of the predicate `isReadyToBeActivated`.

Invariant dependency

An invariant dependency, often called *per-condition*, indicates a necessary requirement for the execution of the *nominal behaviour* of a skill instance s. To

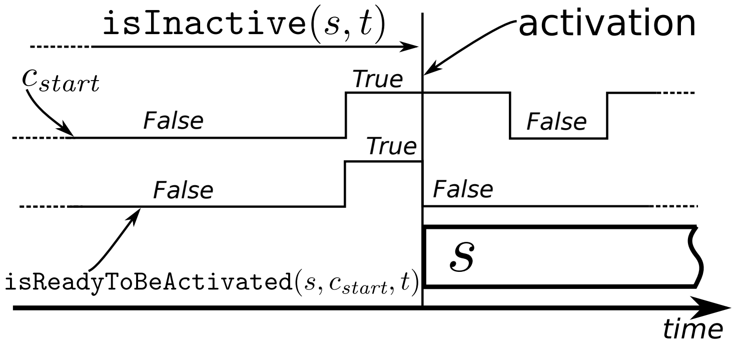


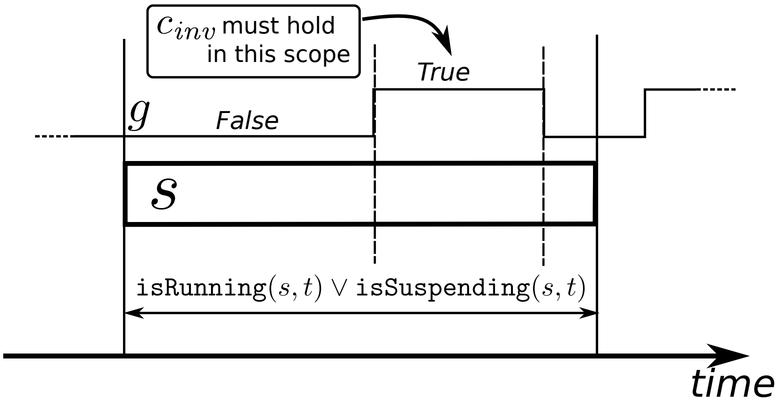
Figure 2.5: The role of the predicate `isReadyToBeActivated`, which lies in the scope prior to the activation of the skill s. In this scope, `isReadyToBeActivated` holds with the condition c_{start}, and the execution engine evaluates the activation of s. After the activation of s the condition c_{start} may vary, but it does not affect the skill execution.

this end, the predicate `invariant` is introduced to bind a user-defined condition c_{inv} that must hold during the execution of s. Due to the previous definition, the invariant relationship lives within the scope of the skill execution; outside that scope, the `invariant` predicate is assumed hold as a convention.

In propositional logic, an invariant is defined by a tuple $<s, c_{inv}, g>$, where s is skill, c_{inv} is a required condition and g is a third condition called *guard condition*:

$$\begin{aligned}\texttt{invariant}(s, c_{inv}, g, t) \leftarrow\ & \underbrace{\big((\neg\texttt{isRunning}(s,t)) \wedge (\neg\texttt{isSuspending}(s,t)\big)}_{scope} \vee \\ & \Big(\underbrace{\big((\texttt{isRunning}(s,t) \vee \texttt{isSuspending}(s,t)\big)}_{scope} \wedge \\ & \underbrace{\big((\neg\texttt{holdsAt}(g,t)) \vee (\texttt{holdsAt}(g,t) \wedge \texttt{holdsAt}(c_{inv},t))\big)}_{guard\ condition}\Big)\end{aligned} \tag{2.6}$$

The role of the guard condition g is to define a continuous (sub-)scope within the execution of s: the condition c_{inv} must hold if also g holds, otherwise the truth value of c_{inv} is not relevant. This allows to express a *per-condition* constraint anchored to the continuous domain, and not discretely on the overall execution of the skill. For the sake of clarity, Figure 2.6 depicts a situation that graphically explains the role of the guard condition. As a side note, Eq. 2.6 can



c_{inv} must hold in this scope
True
g
False
s
$\texttt{isRunning}(s,t) \vee \texttt{isSuspending}(s,t)$
time

Figure 2.6: Scope example of an $\texttt{invariant}(s, c_{inv}, g, t)$ predicate, which lies during the execution of the skill s and depends on the guard condition g. The invariant dependency expresses that the condition c_{inv} must hold within that scope.

be written even without g, but delegating to the boolean expression of c_{inv} the definition of scope; however, the proposed formulation is preferred since it is *explicit*. Finally, multiple invariant dependencies can be applied on the same skill instance s.

An invariant constraint is a *logical proposition*, thus it resides in the symbolic domain. This symbolic constraint can be linked to the concept of geometric or motion invariant [102, 122, 35], which resides in the continuous domain.

2.3.2 The *SDG* Structural Model

This section presents the structural model of the *SDG* and its graphical representation (see Figure 2.7), bringing together all the elements previously described. The first primitive of this model is the **skill** that, when executed, controls the status of its *internal* **condition**s (the second primitive of the *SDG*). In turn, the execution of a skill is driven by the satisfaction of some **dependencies** (third primitive of the *SDG*), which are defined over *external* conditions of the skill. This allows to "connect" the skills between each other, through condition-based dependencies. Let $\mathbb{S}$, $\mathbb{C}$ and $\mathbb{D}$ be, respectively, the sets of skills, conditions and dependencies, then a *SDG* model is defined as a tuple $<\mathbb{S}, \mathbb{C}, \mathbb{D}>$ that describes a *hierarchical hypergraph*. The hierarchy is inherent in the skill model described in Section 2.2. The edge is n-ary relationship that represents a constraint, which can be defined on multiple conditions, and it can influence multiple skill executions.

The mostly adequate structure to describe the *SDG* is based on a *Component-Port-Connector* paradigm, namely the *NPC4* DSL that is discussed in Chapter 6. This section does not provide detailed insights on the *NPC4*, but only on the primitives required for the understanding of the *SDG* model, depicted in Figure 2.7:

- a **node** represents a skill;
- a **port** denotes a *"source"* of a condition, that can be used to define a dependency. A port belongs to one and only one node, exposing an internal condition of a skill s, whether it is an *intended effect* or a *side-effect*;
- a **connector** represents a dependency. Since a well-formed dependency is a relationship between a condition and a skill, the connector connects at least two nodes; one is the subject of the constraint (a skill), the other represents the source of the condition. In addition, connections are *directed*: the target indicates the constrained skill.

Resuming, the structural model of the *SDG* does not allow to connect skills to each other directly, but always through a well-formed condition-based dependency. These connections form a directed and acyclic graph, thus the same skill instance cannot be executed twice; however, multiple instances of the same type can be deployed in the same graph. The previous description defines the structural constraints of a well-formed *SDG* plan.

In the *SDG* model, a skill is a grounded symbolic action that represents a world-transformation, but not all world-transformations are controllable. As a consequence, not all the skills are **executable**:

- an *executable skill* is a skill fully grounded and under control of the execution engine, that is, by executing such a skill it is possible to modify the truth value of its intended effect;
- a skill is *non-executable* in the following cases:
  - it is not fully grounded, *i.e.*, the information available does not suffice to perform the modelled behavior; an example is a skill that grounds the action of grasping an object, but the position of such an object is unknown;
  - the skill represents an action that is not performed by the controlled robot, but by another actor, whether it is a human or a robot.

The above-mentioned is a run-time property of a skill, but it shows the importance of a complete representation of the plan. A well-formed *SDG* model

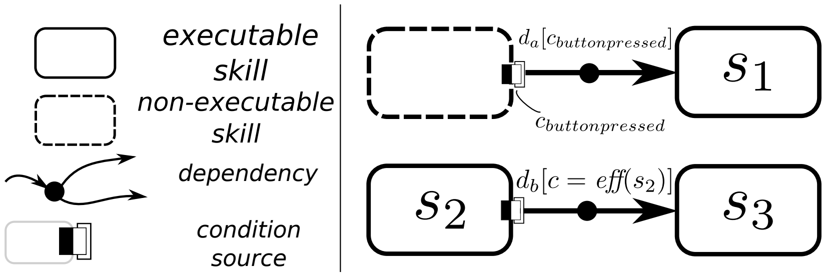


Figure 2.7: Graphical representation examples of structural *SDG* models. On the left, a legend of the graphical elements: executable skills are represented by rounded box with a solid line; non-executable skills are rounded, dashed boxes; a dependency is a connector-based hyperedge; a port exposes internal conditions of a skill. The *SDG* on top-right shows a skill s_1 that has a dependency relationship with the condition $c_{buttonpressed}$. Since the cause of the condition is unknown (that is, another agent will produce such a world-transformation, *e.g.*, a human or a robot), a non-executable skill is deployed. The *SDG* on bottom-right shows two nodes, s_2 and s_3, connected through a dependency constraint d_b. The indirect relationship between the two skills exists since the definition of d_b involves the effect of s_2, $eff(s_2)$.

is complete[6], and it allows to specify the dependency on unknown facts or events, which can be specified later, during the execution of the whole plan. This important feature of the *SDG* approach is called *"Just-in-Time"* and it is discussed in Chapter 5.

Figure 2.7 shows two basic examples. In the first case, a skill s_1 has a dependency constraint d_a, e.g., on activation, against an external condition $c_{button\_pressed}$. Since the source of such a condition is unknown, a non-executable skill is deployed to represent a generic action that modifies the monitored condition. Such an action can be executed by a human or an agent, *e.g.*, *push a button*, thus the skill does not ground any behaviour to be realised. However, during the execution, a run-time update can inform that the robot itself must perform such an action, *e.g.*, by performing the *push a button* action. The second case is trivial, since a skill s_3 is constrained to a dependency d_b. The definition of d_b involves a constraint c, that happens to be the intended effect of a modelled skill s_2 ($eff(s_2)$). As a remark, the $eff(s_2)$ is *internal* with respect to s_2, but *external* with respect to s_3. Directivity in the connections is then explained.

[6]No "floating" ports are allowed, since they must belong to one node. That is, a node representing a skill is required, whether executable or not.

2.4 SDG Executive

This section introduces the SDG-*Executive* (*SDG*-E), the execution engine responsible for the interpretation of a *SDG* model. The *SDG*-E is a software entity that implements the run-time functionalities and policies that activate a *SDG* model, among which:

- skill behaviour selection: a skill models multiple behaviours (see Section 2.2), but only one is deployed at run-time. The *SDG*-E is responsible for activating the skill instances, deploying one behaviour, as well as to switch discretely from a behaviour to another during the skill execution, in accordance with the satisfaction of the modelled dependencies. Therefore, the *SDG*-E executes the transitions between skill `status`, which are discussed in details in the next sub-section;
- *Execution Monitoring*: the *SDG*-E enables the monitoring of interesting facts, whose monitor functions that bind to logical conditions that may trigger a change in the deployed behaviour. In other words, not all logical conditions described in a *SDG* model are monitored, but only those that are relevant in the *run-time context*;
- online composition of a *SDG* model: due to a failure or an external event, a *SDG* plan may vary at run-time. The *SDG*-E implements the functionalities that allow to dynamically (re-)compose an existing *SDG* model. Further details on this topic are available in Chapter 4 and Chapter 5.

Therefore, the *SDG*-E implements a **scheduling** strategy of a given *SDG* model; it does not concern about *which* skills are planned, but it is responsible for determining *when* a certain skill is executed.

2.4.1 Skill Execution Status Diagram

This section introduces the *Skill Execution Status Diagram* (*SESD*) that defines which transitions are available between the behaviours modelled in a skill instance s (see Figure 2.8). These transitions are enabled based on the satisfaction of dependencies that constrain the skill instance. Thus, an online change on a monitored condition can trigger a transition from one skill `status` to another, determining whether a skill is executed, it is failed or its nominal behaviour is replaced by a non-nominal behaviour. The notation $\longmapsto$ expresses a *transition*; as an example, $s.\texttt{inactive} \longmapsto s.\texttt{running}$ refers to a *transition*

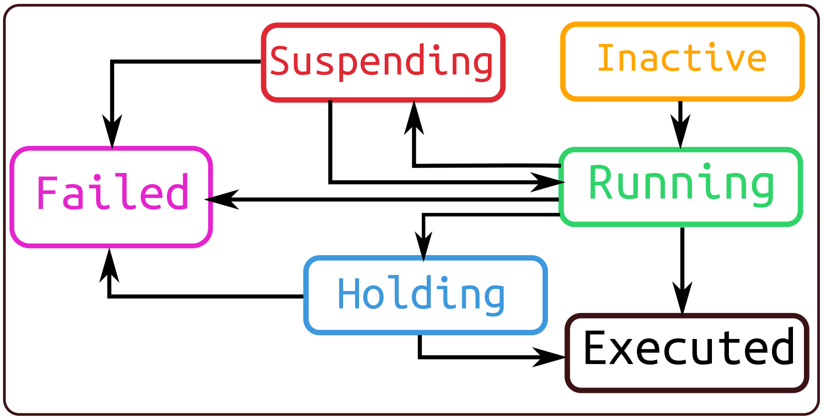


Figure 2.8: Skill Execution Status Diagram (*SESD*). This diagram describes the transitions available between skill status. `inactive` is initial status of a skill instance. `executed` and `failed` are terminal status that indicate a successful execution or a failure, respectively. `running` status represents the nominal behaviour described by the skill, while `suspending` describes the non-nominal behaviour. Finally, `holding` allows to maintain the desired *effect* obtained in `running` status.

of s, from `inactive` to `running`. A detailed description in propositional logic flavor is reported below.

Initial and Terminal Status

For all the skill instances in a *SDG* model, the initial status is `inactive`. After being executed, a skill may succeed or not: the terminal status `executed` and `failed` represents this semantic difference.

Activation, $s.\texttt{inactive} \longmapsto S.\texttt{running}$

The activation of a skill is strictly related to the dependency constraint `isReadyToBeActivated` illustrated in Section 2.3.1. As soon as the condition c_{start} holds, the *SDG*-E switches the skill status to `running`, activating the related behaviour. Formally, let s be a node in `inactive` status, that is $s.\texttt{inactive}$, then the transition $s.\texttt{inactive} \longmapsto s.\texttt{running}$ is enabled if the predicate $\texttt{isReadyToBeActivated}(s, c_{start}, t)$ holds:

$$\texttt{isRunning}(s, t^+) \leftarrow \texttt{isReadyToBeActivated}(s, c_{start}, t). \tag{2.7}$$

Invariant constraint, $s.\texttt{running} \longmapsto s.\texttt{suspending}$ and $s.\texttt{suspending} \longmapsto s.\texttt{running}$

The transitions between the nominal execution and the non-nominal execution of a skill instance s are determined by the invariant constraints presented in Section 2.3.1. Unlike the activation dependency, a skill instance can be subject to multiple invariant constraints. The violation of one of the invariant constraint is *sufficient* to trigger a transition from the `running` status (nominal behaviour) to a `suspending` status (non-nominal behaviour). The nominal execution is restored only if all the invariants are satisfied again. Let $\mathbb{I}_s$ be a set of the invariants applied to s, $\mathbb{I}_s = \{\texttt{<}(s, c_{inv_1}, g_1)\texttt{>}, \texttt{<}(s, c_{inv_2}, g_2)\texttt{>}, \dots\}$, then

$$s.\texttt{suspending} \leftrightarrow \exists i \in \mathbb{I}_s : \neg\texttt{invariant}(i, t), \tag{2.8}$$

$$s.\texttt{running} \rightarrow \forall i \in \mathbb{I}_s : \texttt{invariant}(i, t). \tag{2.9}$$

Moreover, if the skill instance s is a composite and its status is `suspending`, then all the children skills s_l in `running` status are suspended:

$$\begin{aligned} \texttt{isSuspending}(s, t) \rightarrow \forall s_l,\ \texttt{isParent}(s, s_l) \wedge \\ \texttt{isRunning}(s_l, t^-) \wedge \texttt{isSuspending}(s_l, t). \end{aligned} \tag{2.10}$$

Successfully Termination, $s.\texttt{running} \longmapsto s.\texttt{executed}$ and $s.\texttt{running} \longmapsto s.\texttt{holding}$

The nominal execution of a skill s_k succeeds if the intended effect $c_{eff} = eff(s_k)$ is achieved. In this case, the skill can be dismissed only if there are no dependencies in the *SDG* model based on the intended effect $eff(s_k)$ that constrain another skill ($s_j \in \mathbb{S}$), which is under execution or not activated yet. Otherwise, the status of the skill s_k switches to `holding`, that is the behaviour that ensures the c_{eff} to hold over the time.

In propositional logic,

$$\begin{aligned}
&\exists s_j \in \mathbb{S},\ j \neq k,\ c_{\mathit{eff}} = \mathit{eff}(s_k),\ \mathbb{I}_{s_j} = \{<(s_j, c_{inv_1}, g_1)>, \dots\},\\
&\quad \big(\texttt{isRunning}(s_k, t) \to \textbf{True}\big),\\
&\quad \big(\neg\texttt{isExecuted}(s_j, t) \land \neg\texttt{isHolding}(s_j, t) \land \neg\texttt{isFailed}(s_j, t)\big) \to \textbf{True},\\
&\quad \texttt{isHolding}(s_k, t^{+}) \leftarrow\\
&\quad \Big((\texttt{isReadyToBeActivated}(s_j, t) \to \textbf{False}) \leftarrow \big((\neg\texttt{holdsAt}(c_{\mathit{eff}}, t))\big)\Big) \lor\\
&\quad \Big(\forall i \in \mathbb{I}_{s_j},\ (\texttt{invariant}(i, t) \to \textbf{False}) \leftarrow (\neg\texttt{holdsAt}(c_{\mathit{eff}}, t))\Big).
\end{aligned} \tag{2.11}$$

In short, a `running` skill terminates once its intended world-transformation is obtained, and no other skill requires such a transformation to persist. If the skill s_k is composite and in `holding` status, then those skill instances s_l such that $\texttt{isParent}(s_k, s_l)$ holds and their intended effect contribute in the definition of $\texttt{eff}(s_k)$ are also in `holding` status:

$$\forall s_l\ : (\texttt{isParent}(s_k, s_l) \to \textbf{True}),$$

$$(\texttt{holdsAt}(\texttt{eff}(s_k), t) \to \textbf{False}) \leftarrow (\neg\texttt{holdsAt}(\texttt{eff}(s_l), t)), \tag{2.12}$$

$$\texttt{isHolding}(s_l, t) \leftrightarrow \texttt{isHolding}(s_k, t). \tag{2.13}$$

Execution Failure, $s.\texttt{running}, s.\texttt{suspending}, s.\texttt{holding} \longmapsto s.\texttt{failed}$

The execution of any behaviour modelled in the skill can fail. As discussed in Section 2.2.5, multiple failure conditions exist, and only one is *sufficient* to trigger a transition to a `failed` status:

$$\begin{aligned}
&\texttt{isFailed}(s, t^{+}) \leftarrow\\
&\qquad \exists c_{s,k} = \mathit{fail}(s, k) : \texttt{holdsAt}(c_{s,k}, t) \land\\
&\qquad (\texttt{isRunning}(s, t) \lor \texttt{isHolding}(s, t) \lor \texttt{isSuspending}(s, t)).
\end{aligned} \tag{2.14}$$

Moreover, if the skill instance s_k is composite, and at least one child s_l is in `failed` status, then also s_k fails:

$$\texttt{isFailed}(s_k, t) \leftarrow \exists s_l,\ \texttt{isParent}(s_k, s_l) \land \texttt{isFailed}(s_l, t). \tag{2.15}$$

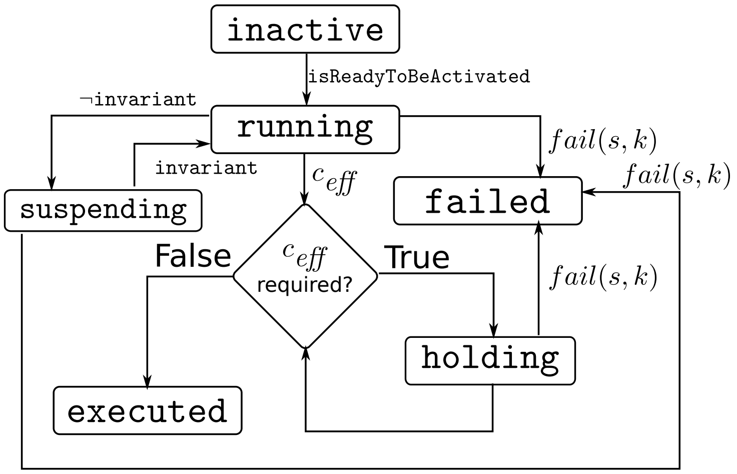


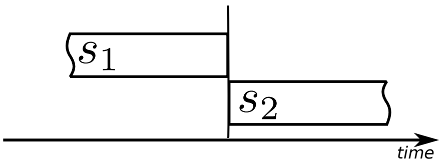
Figure 2.9: Simplified overview of the algorithm adopted for the implementation of the *SDG*-E, in a dataflow form. The predicates that influence the transitions have been briefly reported in an informal fashion.

To conclude, Figure 2.9 reports a simplified overview of the *SDG*-E algorithm that manages each skill.

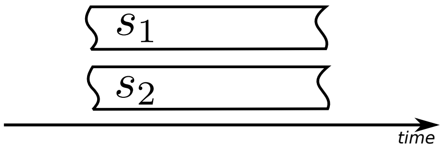
2.5 The micro Skill Dependency Language (uSDL)

This section introduces the micro Skill Dependency Language (*uSDL*), a minimalistic language to describe relationships between skills and conditions. The language aims to build a *SDG* model through a set of declarative rules that describes an executable situation (plan). In fact, the `invariant` dependency (see Section 2.3.1) is rather generic, and the *uSDL* limits the definition of `invariant` dependency to a subset of interesting use-cases shown in Figure 2.10. Concretely, the *uSDL* aims to express explicitly declarative constraints that determine the following situations between a pair of skills:

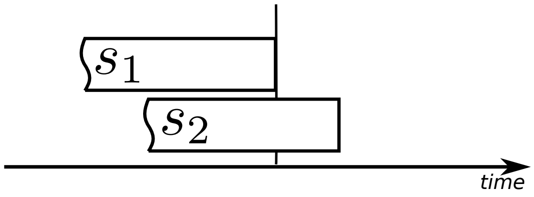
- **Strict sequence**, the activation of the execution of a skill instance s_2 depends on the successful execution of a skill instance s_1 (see Figure 2.10a);
- **Pure concurrency**, there is no relationship between the execution of the skill instances s_1 and s_2, that is, the skills can be executed as parallel activities shown in Figure 2.10b;



(a) Strict sequence between the skills s_1 and s_2, as soon as s_1 terminates, s_2 is activated.



(b) Pure concurrency example between s_1 and s_2, since there is no relationship between the skills.



(c) Conditional concurrency between s_1 and s_2: s_1 and s_2 are executed concurrently, but s_1 must terminate earlier than s_2.

Figure 2.10: Situation examples of *strict sequence* (Figure 2.10a), *pure concurrency* (Figure 2.10b) and *conditional concurrency* (Figure 2.10c) between a pair of skills s_1, s_2.

- **Conditional concurrency** can be seen as a relaxed version of a strict sequence, as well as a constrained version of pure concurrency. In this situation, the skills s_1 and s_2 can run concurrently, since there is no activation dependency between them. However, the intended effect of one skill can influence the execution of the other, causing a change on the deployed behaviour. Therefore, a relationship between the couple of skills exists. An example is depicted in Figure 2.10c, where a skill s_1 must terminate successfully within the scope of the execution of s_2.

In addition, the *uSDL* covers explicitly the concept of **common-sense law of inertia**. Such a concept has been firstly derived from the *frame problem* [144] caused by a *world model closure*. In literature, a common assumption is to consider the effect achieved by the execution of an action as *persistent*, that is, the effect holds is no other actions are applied to influence its truth value. Examples of this assumption are: a glass placed on a table stays on the table, a door that has been opened stays open, a grasped object is still in the robot hands if no action is taken to release such an object. However, in a dynamic environment this assumption is not always valid, requiring a persistent action to maintain the achieved effect, *e.g.*, a grasped object is still in the robot hands if the grasping persists. Therefore, the common-sense law of inertia describes a situation where a skill must hold the achieved effect, and the symbolic inertia is transformed into a constraint between the skill that realise such an effect, and those that depends on its truth value for their execution.

As explained in Section 2.7, *strict sequence* and *pure concurrency* are already covered by the state-of-the-art. The contributions of the *uSDL* are the *conditional concurrency* and the *common-sense law of inertia*, the two primary design drivers of the language.

Another relevant design driver is the requirement of independent and semantically consistent declarative approach by means of a set of rules that can be added or removed also at run-time.

The proposed language is developed as a *textual* model, but graphical elements are also used for the sake of clarity. There is no strong emphasis on the syntax choice, which can be easily adapted in accordance with the implementation tools. Where possible a polish notation is preferred, since it allows to build an *Abstract Syntax Tree* (AST) that facilitates both parser and lexer processes.

2.5.1 The Language

The core of the language are the **declarative rules** `toStart`, `continuesIf` and `latches` that constrain a skill instance to a condition, defining a connection in the *SDG* model between a node (the skill) and a port (the condition). Therefore, the primitives of the language are: `Skill`, `Condition` and the **dependencies** created by the declarative rules. `Condition`s are symbolic, and they are defined as a boolean expression or as a mapping to the continuous domain through monitor functions: a `monitor` relationship serves this purpose. The relationships `is-effect` and `is-side-effect` denote if a condition is *intended effect* or *side-effect* of a skill, respectively. Another relationship is the `contains` that indicates the hierarchy between skills. Lastly, *uSDL* primitives and relationships have an *Unique Identifier Number* (UID), allowing introspection and online manipulation of their properties.

Identity
Identity is given to each primitive instance by simple declaration:

$$\texttt{Skill:}\;\; \texttt{s1, s2,...} \tag{2.16}$$

$$\texttt{Condition:}\;\; \texttt{c1, c2,...} \tag{2.17}$$

Dependency primitives are declared through the declarative rules `toStart`, `continuesIf` and `latches` as described below.

`is-effect-of, is-side-effect-of,eff`
By default, each skill instance has an intended effect, thus the declaration of a skill instance `s` also deploys a `Condition c` such that `is-effect-of(s,c)`. Moreover, the intended effect can be retrieved with the `eff(s)`. Side-effects must be explicitly declared with `is-side-effect-of`, for example:

$$\texttt{is-side-effect-of(s1,c1)} \tag{2.18}$$

states that the condition `c1` is side effect of the skill `s1`.

`monitor`
Each condition binds a monitor function for its evaluation, or to a boolean expression, for instance

$$\texttt{Condition:\ \ c1,c2,c3}$$

$$\texttt{monitor(c1,and(c2,c3))} \tag{2.19}$$

represent the following logical proposition

$$\texttt{holdsAt}(c1,t) \leftarrow \texttt{holdsAt}(c2,t) \wedge \texttt{holdsAt}(c3,t). \tag{2.20}$$

If the monitor function is defined in the continuous domain, a symbolic function identifier denotes which monitor function implementation binds to the target `Condition`:

$$\texttt{monitor(c1,'fncmon42').} \tag{2.21}$$

It is *SDG*-E role to bind that `Condition` to the underlying monitor function implementation, and to manage the `Condition` evaluation online. The implementation of these is discussed in Chapter 3, and further examples are in Chapter 4.

`contains`
The relationship `contains` conforms to the skill model illustrated in Section 2.2, and it denotes the hierarchy between skills, thus:

$$\texttt{contains(s1,s2)} \tag{2.22}$$

expresses that the skill $s1$ contains the skill $s2$.

The *uSDL* has three types of dependency relationships to add constraints to the skill execution: `toStart`, `continuesIf` and `latches`. Each of these bind a skill with conditions, and it returns a dependency primitive, for instance

$$\texttt{d1 = toStart(s,c).}$$

If not captured, the dependency primitive can be retrieved inspecting the connectivity of the structural *SDG* model.

`toStart`

The `toStart` relationship represents a necessary but not sufficient condition for the activation of a skill s.

`toStart`(s, c) reads as *"to start the skill s, the condition c is required to hold".*

Formally, the `toStart` is nothing more than syntactic sugar that composes the activation condition c_{start} of s (see predicate `isReadyToBeActivated` in Eq. 2.5). Let $\mathbb{A}_s$ be the set of all the conditions involved in a `toStart` constraint of the skill s, then the truth value of c_{start} is subject to:

$$\texttt{holdsAt}(c_{start}, t) \leftarrow \forall c_j \in \mathbb{A}_s : \texttt{holdsAt}(c_j, t). \tag{2.23}$$

Furthermore, if $\mathbb{A}_s$ is *empty set* ($\mathbb{A}_s = \{\emptyset\}$), then c_{start} always holds.

`continuesIf`

The `continuesIf` relationship implements an event-based guard over the nominal execution of s.

`continuesIf`$(s, c_1, c_{s,n})$ reads as *"the execution of skill s continues if c_1 holds when a rising event e defined over $c_{s,n}$ occurs".*

This relationship is a specific case of an `invariant` constraint, where the condition arguments bind the following predicates of the invariant tuple:

- c_{inv}: is directly linked to c_1, as an external condition that must hold along s execution scope;
- g **guard condition** predicate is defined as an occurrence of a rising event over the condition $c_{s,n}$. Moreover, $c_{s,n}$ must be a side-effect of s (n subscript indicates to which side-effect refers): this constraint is fundamental in the `continuesIf` definition, since it binds the scope of the `invariant` dependency on the execution of s itself. In propositional logic,

$$\begin{aligned}\texttt{holdsAt}(g, t) \leftarrow \\ & c_{s,n} = \texttt{side-eff}(s, n),\ \exists e_{c_{s,n},j} \in \texttt{events}(c_{s,n}) \\ & : \texttt{occurs}(e_{c_{s,n},j}, t) \wedge \texttt{eventRising}(e_{c_{s,n},j}, t).\end{aligned}$$

This "point in time"[7] rule is fundamental to implement a *conditional concurrency* relationship between a pair of skills. Considering a skill pair s_1,s_2, the guard g is defined over the behaviour of s_2, while the skill coupling is given by c_1, where $c_1 = \texttt{eff}(s_1)$. Concretely, such a mechanism permits to define statements as *"s_1 must terminate successfully before s_2 terminates"*: the condition $c_{s,n}$ expresses how close is the execution of s_2 to its termination. Similarly to the `toStart`, multiple `continuesIf` relationships can be applied to the same skill s; some examples are shown in Section 2.5.2.

`latches`

The `latches` relationship implements the *common-sense law of inertia*, expressed as a persistency requirement of an external condition c in the scope of the skill execution s. Moreover, such a requirement is not required along the overall execution scope, but only after that the condition c holds for the first time. That is, c may never hold in the scope of the execution of s, and still such a constraint is not violated; instead it is violated if the truth value of c holds, and later it does not. In short, the scope of the execution s *latches* c to hold. This relationship is mostly used in combination with a `toStart` or a `continuesIf`, and examples of usage are shown in Section 2.5.2.

> **`latches`**(s, c) reads as *"the nominal execution of s latches the truth value of c, as soon as c condition holds within s scope.*

The relationship indicates that the skill s *requires* the condition c; the previous does not imply that s behaviour is actively maintaining that condition, since c is external to s. However, the condition c is possibly maintained by another skill for which c is internal (*i.e.*, the intended effect).

The `latches` defines an `invariant` constraint having a guard predicate based on c_{inv}:

$$\begin{aligned}&\texttt{holdsAt}(g,t) \leftarrow \\ &\quad \Big(\texttt{first-occur}(c_{inv},t) \wedge \\ &\quad \underbrace{(\texttt{isRunning}(s,t) \vee \texttt{isSuspending}(s,t) \vee \texttt{isReadyToBeActivated}(s,t))}_{scope}\Big),\end{aligned} \tag{2.24}$$

[7]The guard condition of the `continuesIf` rule is event-based that holds only at one point in time.

where the scope term contains the `isReadyToBeActivated` predicate, such that the occurrence is also captured in the neighbourhood of the activation. The predicate `first-occur` captures the first occurrence of c_{inv} and it can be written as:

$$\begin{aligned} &\texttt{first-occur}(c_{inv}, t) \leftarrow \\ &\qquad \exists e_{c,j} \in events(c_{inv}) : \\ &\qquad \texttt{happened}(e_{c,j}, t) \wedge \texttt{eventRising}(e_{c,j}, t). \end{aligned} \tag{2.25}$$

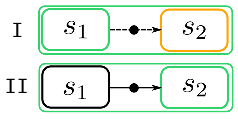
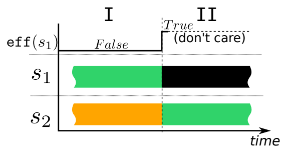
2.5.2 Common Situations

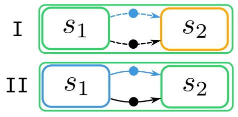
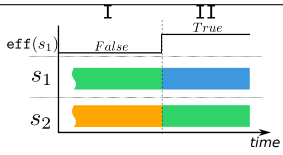
The *uSDL* expresses execution dependencies between a set of skills. Table 2.3 and Table 2.4 resume some run-time examples of *strict sequence*, *pure concurrency*, *conditional concurrency*. In detail:

- **strict sequence** is shown as a first example; s_1 has no activation constraints, while s_2 requires $\texttt{eff}(s_1)$ to start. Therefore, s_2 is executed immediately after the termination of s_1;
- **strict sequence, with inertia** is reported as a second case; s_2 requires $\texttt{eff}(s_1)$ for both activation and invariant conditions over the nominal execution of s_2. As a consequence, s_1 switches to `holding` instead of `executed` status;
- **pure concurrency** between a pair is shown as a third example; there are no constraints on both s_1 and s_2, so their execution start immediately and they are independent. In the example, s_2 terminates before s_1, but also the opposite is possible;
- **conditional concurrency** (with inertia) is one of the most interesting cases. In this situation, s_1 and s_2 are not constrained over their activation, but s_2 relies on the effect of s_1 ($\texttt{eff}(s_1)$) to continue its nominal behaviour. The guard condition is determined by one of the side effects of s, $c_{sf} = \texttt{side-eff}(s, j)$. If the `continuesIf` constraint is not satisfied, the nominal execution of s_2 is suspended, so a non-nominal behaviour is performed. As soon as the constraint is satisfied again, the nominal execution is restored.

Whenever the `invariant` constraints are not satisfied, the executive changes the status of the constrained skill to `suspending`. The latter may occur due to a failure during execution of another skill, responsible for one specific condition. Reacting by switching to `suspending` status is fundamental to implement a

Rules	**SDG**	**Outcome** (Gantt Char)

`Skill: s1, s2`
`toStart(s2,eff(s1))`
		

`Skill: s1, s2`
`toStart(s2,eff(s1))`
`latches(s2,eff(s1))`
		

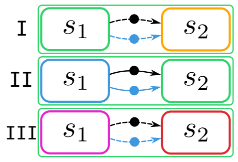
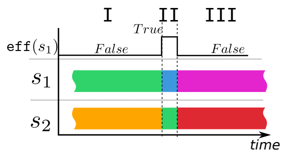
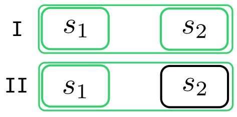
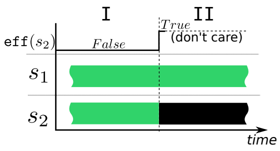
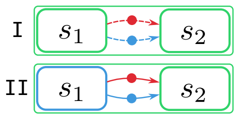
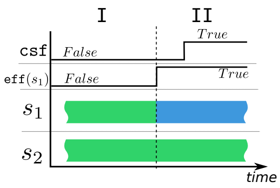
`Skill: s1, s2`
`toStart(s2,eff(s1))`
`latches(s2,eff(s1))`
		

Table 2.3: Execution examples of some common situations. The first column reports the textual *uSDL* model; the middle column shows the resulting *SDG* structure; the last column shows the situation outcome, expressed as a Gantt chart. The *SDG* in the middle column are *snapshots* of the run-time execution. Some structural primitives (e.g., ports) have been omitted. The color code of the nodes reflects the online skill status: `running`(green), `executed`(black), `holding`(blue), `inactive`(yellow), `suspending`(red), `failed`(purple). The color code of connectors represents the dependency type: `toStart`(black), `continuesIf`(red), `latches`(blue). The run-time information over the constraint (satisfied or not) is expressed by the shape of the line: "holds" is a solid line, "does not hold" is a dashed line. For the sake of clarity, port elements over the structural *SDG* model are not explicitly represented. The *SDG* snapshots represent the overall status along one period of time, as shown in the outcome column. Each row presents a different situation, in order: *i)* strict sequence; *ii)* strict sequence, with inertia; *iii)* same of *ii)*, but with a run-time failure. Pure concurrency and conditional concurrency cases are shown in Table 2.4.

behaviour that prevents the scheduling of undesired motions. As a major feature of the proposed approach, the *SDG* provides some information to reason about the action failure: which skill failed, in which composition and in which context (*e.g.*, the dependencies that are not respected). In case of a failure, a high-level planner can exploit these information and provide a better plan than the original one.

Rules	**SDG**	**Outcome** (Gantt Char)
`Skill: s1, s2`		

`Skill: s1, s2`
`Condition: csf`
`is-side-effect-of(s2,csf)`
`latches(s2, eff(s1))`
`continuesIf(s2,`
`,eff(s1), csf)`
		

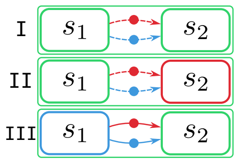
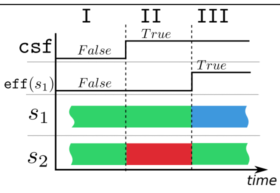
		

Table 2.4: Execution examples of pure concurrency and conditional concurrency, in order: *i)* pure concurrency; *ii)* conditional concurrency with inertia; *iii)* same *SDG* model of *ii)*, but s_2 goes to `suspending` status due to the intended effect of s_1 not achieved yet. This table follows the same legend expressed in Table 2.3.

2.5.3 Example

Listing 2.3 reports the *uSDL* textual model of a toy example, hierarchical composition included. The resulting structural model is graphically shown in Figure 2.11.

```
Skill: s1,s2,s3,sA,sB
Condition: side_eff_sb

is-side-effect-of(sB,side_eff_sb)
contains(sA,s1)
contains(sA,s2)
contains(sA,s3)

d1=toStart(s2,eff(s1))
d2=toStart(s3,eff(s1))
d3=latches(s3,eff(s1))
d4=latches(sB,eff(sA))
d5=continuesIf(sB,eff(sA),side_eff_sb)

monitor(eff(sA),and(eff(s2),eff(s3))
monitor(side_eff_sb,"fncmon425")
monitor(eff(s1),"fncmon12")
monitor(eff(s2),"fncmon325")
monitor(eff(s3),"fncmon247")
monitor(eff(sB),"fncmon543")
```

Listing 2.3: Code snippet of an *uSDL* example.

Note that *uSDL* is a minimal language, as a compromise between completeness and verbosity of the model. In fact, it limits the expressivity of the `invariant` constraints to only two classes (*e.g.*, `latches` and `continuesIf`); nevertheless this suffice to describe the most common situations. A full model description (explicit connections, fully coupling between skills) is provided in the Listing 2.4. Such a model is encoded as a JSON object (*JavaScript Object Notation Data Interchange Format*, see [28]). The JSON allows to describe an independent model with respect to an underlying, language-dependent implementation of the *SDG*-E. Furthermore, the above-mentioned model conforms to the (M2) meta-model described in JSON-Schema [57] reported in Appendix A.1.

```
{
"metamodel":"http://people.mech.kuleuven.be/~u0072295/sdg-v01",
  "skills": [
    { "id": "example", "type": "skill" },
    { "id": "s1", "type": "skill", "eff": "eff_s1" },
    { "id": "s2", "type": "skill", "eff": "eff_s2" },
    { "id": "s3", "type": "skill", "eff": "eff_s3" },
    { "id": "sA", "type": "skill", "eff": "eff_sA",
      "side-eff":[ "eff_s2", "eff_s3" ] },
    { "id": "sB", "type": "skill", "eff": "eff_sB",
      "side-eff":[ "side_eff_sb" ] }
  ],
  "contains" : [
    { "parent"   : "example",
      "children" : [ "sA", "sB" ]
    },
    { "parent"   : "sA",
      "children" : [ "s1", "s2", "s3" ]
    }
  ],
  "conditions" : [
   { "id" : "eff_s1", "type": "condition",
     "monitor": { "type" : "function", "id" : "fncmon12" }
   },
   { "id" : "eff_s2", "type": "condition",
     "monitor": { "type" : "function", "id" : "fncmon325" }
   },
   { "id" : "eff_s3", "type": "condition",
     "monitor": { "type" : "function", "id" : "fncmon247" }
   },
   { "id" : "eff_sA", "type": "condition",
     "monitor": { "type" : "expr",
     "expr":{ "operator": "and", "args": ["eff_s2", "eff_s3"]}}
   },
   { "id" : "eff_sB", "type": "condition",
     "monitor": { "type" : "function", "id" : "fncmon543" }
   },
   { "id" : "side_eff_sb", "type": "condition",
     "monitor": { "type" : "function", "id" : "fncmon425" }
   }
  ],
  "dependencies" : [
    { "id" : "D1", "type" : "dependency",
      "relationship" : "toStart", "condition": "eff_s1",
      "requiredby" : [ "s2", "s3" ]
    },
    { "id" : "D2", "type" : "dependency",
      "relationship": "latches", "condition": "eff_s1",
      "requiredby" : [ "s2" ]
    },
    { "id" : "D3", "type" : "dependency",
      "relationship": "latches", "condition": "eff_sA",
      "requiredby" : [ "sB" ]
    },
    { "id" : "D4", "type" : "dependency",
      "relationship": "continuesIf", "condition": "eff_sA",
      "guard" : "side_eff_sb",
      "requiredby" : [ "sB" ]
    } ]
}
```

Listing 2.4: Formal structural model of the *SDG* in Figure 2.11, in JSON format

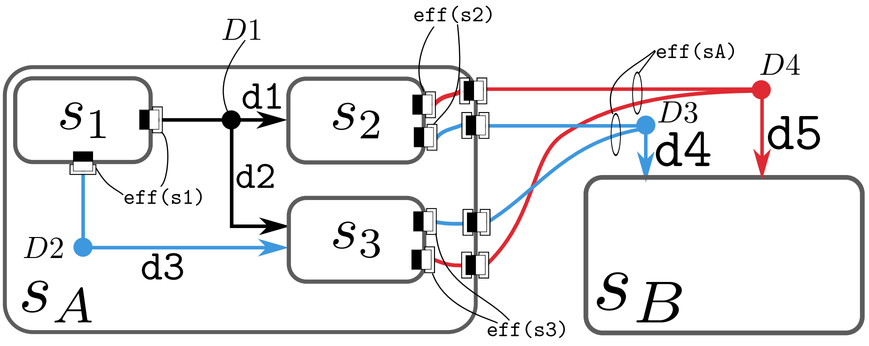


Figure 2.11: Structural model of a complete and well-formed toy example reported in Listing 2.4. Only static information is reported: nodes do not assume a status (grey color). The skill s_A exposes externally the intended effect of s_2 and s_3 as side-effects of s_A. Furthermore, the intended effect of the composite skill s_A is achieved when both intended effects of s_2 and s_3 hold. Thus, the dependency constraints $d4$ and $d5$ over the effect of s_A are expanded to s_A internals. $D1$–$D4$ refers to the connectors that are represented in the JSON model in Listing 2.4.

2.6 Open a Drawer Scenario

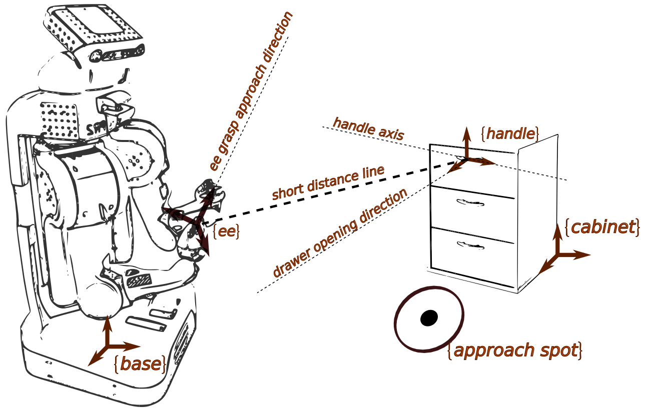


Figure 2.12: Open a Drawer Scenario. The picture shows the geometric entities used to define the motion skills (see Table 2.5).

Finally, this section applies the proposed *SDG* formalism and the related *uSDL* to model a concrete use-case. Recalling from Chapter 1, the aim is to show the benefits of the proposed approach with respect to the limitations discussed on the *Open a Drawer Scenario*.

Figure 2.12 shows the layout of frames and geometric entities involved to describe the motion skills, briefly resumed in Table 2.5. The skills are implemented as a configuration of a Constraint Optimization Problem (COP). Thus, each skill in Table 2.5 is formulated as a set of constraints over relationships of geometric entities. COP solution generates the instantaneous robot motion that satisfies the imposed constraints. That is, the composability of the COP approach allows to fulfill multiple skill executions concurrently, whenever those are not conflicting within the tolerance allowed by the monitor functions. It is beyond the scope of this section to provide further details about the motion specification; a concrete contribution will be discussed in Chapter 3.

A concrete set of execution dependencies is specified in a declarative form by constraining the skills of Table 2.5 with the logical relationships of Listing 2.5, and Figure 2.13. This description allows a more detailed specification with respect to common alternatives based on Finite State Machines (FSMs). Tables 2.6 and 2.7 compare two different execution outcomes of the same *uSDL* model, expressed in a Gantt-chart form.

The execution reported in Table 2.6 shows a nominal situation: firstly, the

Skill	Constraint description
`approach_drawer`	The base position of the robot is constrained to reach the approach spot (Figure 2.12).
`align_line`	The gripper is constrained to point toward the handle (i.e. the approach direction must be aligned with the shortest distance line between the end effector and the handle).
`align_rot`	The axis of the end effector perpendicular to the gripper movements must be aligned with the handle axis.
`open_gripper`	The gripper is open so that it can enclose the handle.
`move_to_handle`	The distance between the gripping point and the handle is reduced to zero.
`grasp_handle`	The gripper is closed in order to take hold of the handle.
`pull_drawer`	The end effector moves/exerts force along the drawer opening direction.

Table 2.5: Open a drawer scenario: qualitative and verbose description of the constraint-based motion related to each skill. Some of the constraints are expressed in terms of the geometric features depicted in Figure 2.12).

robot approaches to the drawer; secondly the skills `align_rot`, `align_line` and `open_gripper` prepare the gripper to grasp the handle; then the gripper is moved to the handle position; and finally the handle is grasped and the end effector pulls the drawer. Multiple situations described in Section 2.5.2 can be identified in this execution: `approach_drawer` and `align_rot` skills are executed as *strict sequence*; `align_rot` and `align_line` are executed concurrently with respect to each other; logical inertia is required for the `pull_drawer` skill, since the gripper must grasp the handle *before* the pulling action, but also during the pulling execution.

Another execution that differs from the previous is shown in Table 2.7. In this case, the dynamics of the gripper is slower than expected with respect to the motion that approaches to the handle. If the gripper is not fully open when it reaches the handle, an unexpected collision may occur, preventing the achievement of the intended effect. The latter also holds for the `align_rot` constraint. A feasible solution would be to execute these skills as a *strict sequence*, but such a solution is extremely conservative with respect to the robot capabilities. Therefore, a guard is imposed on the continuous behaviour of `move_to_handle`, that is allowed to `continuesIf` the gripper has been properly prepared for the grasping action before it is *close enough* to the handle. The above description is expressed by the constraints imposed by lines 14 and 15 in

```
Skill: approach_drawer, align_line, align_rot,
    open_gripper, move_to_handle, grasp_handle,
    pull_drawer

Condition: close_to_handle

is-side-effect-of(move_to_handle,close_to_handle)

d1=toStart(align_rot,eff(approach_drawer))
d2=toStart(align_line,eff(approach_drawer))
d3=toStart(open_gripper,eff(approach_drawer))
d4=toStart(move_to_handle,eff(align_line))
d5=latches(move_to_handle,eff(align_rot))
d6=latches(move_to_handle,eff(align_line))
d7=latches(move_to_handle,eff(open_gripper))
d8=continuesIf(move_to_handle,eff(open_gripper),
    close_to_handle)
d9=continuesIf(move_to_handle,eff(align_rot),
    close_to_handle)
d10=toStart(grasp_handle,eff(open_gripper))
d11=toStart(grasp_handle,eff(move_to_handle))
d12=toStart(pull_drawer,eff(grasp_handle))
d13=latches(pull_drawer,eff(grasp_handle))
```

Listing 2.5: ***uSDL*** **dependency constraints applied on the open a drawer scenario. Monitor functions descriptors have been omited.**

Listing 2.5, where `close_to_handle` side effect of `move_to_handle` is evaluated as Euclidean distance between the handle and the gripper. As a consequence, the `move_to_handle` motion is suspended if the logical constraint is not satisfied, and restored later when the dependencies have been satisfied. In this case, the behaviour of the `isSuspending` status is implemented as maintaining the current distance from the handle.

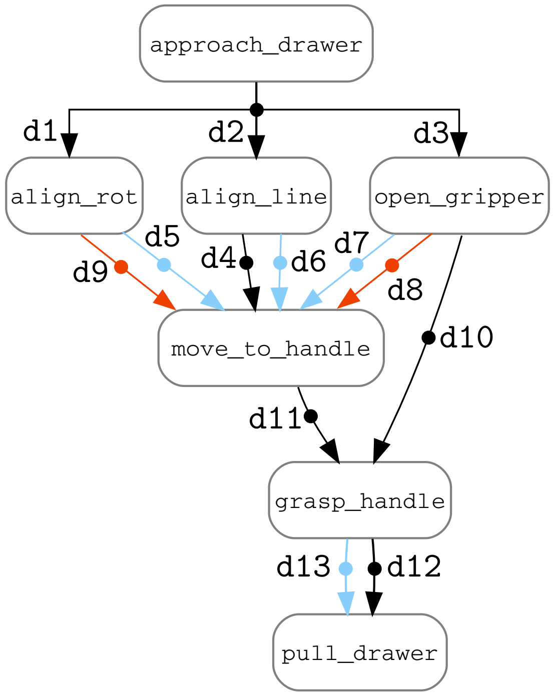


Figure 2.13: *SDG* model of the *Open a Drawer Scenario* described in the *uSDL* in the Listing 2.5. The model illustrated is mainly structural and does not show any online information. For the sake of clarity, some structural primitives (*e.g.*, ports) have been omitted.

The outcomes illustrated in Tables 2.6 and 2.7 have been obtained from experiments conducted in both simulation environment and an experimental setup. Further details on this use-case can be found in [137][8], and multimedia material of the experiments is available online[9].

[8]The work presented in [137] shows early results with respect to the approach discussed in this chapter. Core functionalities and experimental validations are correct and coherent. However, many contributions have been formalised in this chapter, introducing explicitly the *SDG* and the resulting *uSDL*, that slightly differs with respect to the original semantics. For instance, the relationship `toHold` in [137] is equivalent to the `latches`, as well as the `toEnd` that is equivalent to the `continuesIf`.

[9]`https://youtu.be/NzSLYboz2Os`

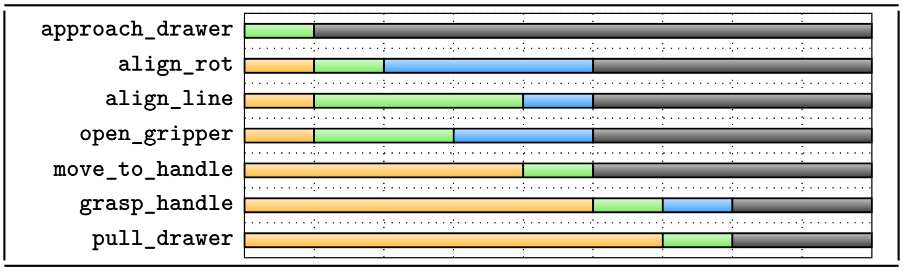


Table 2.6: Nominal execution outcome of the *SDG* model in Figure 2.13, in a Gantt-chart representation. Some motions exhibit a *strict sequence* execution, other run concurrently. All the constraints have been respected during the execution, thus no skill execution fails or switches to non-nominal behaviour.

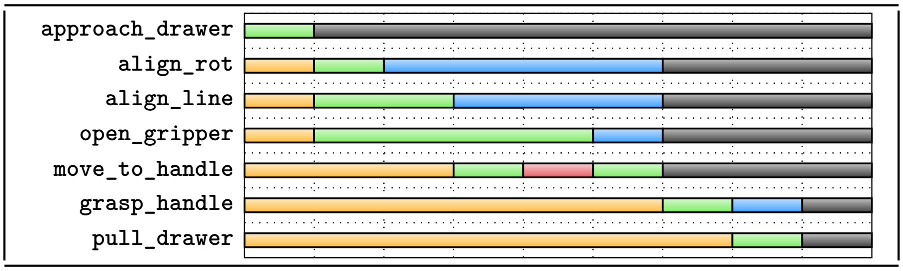


Table 2.7: Non-nominal execution outcome of the *SDG* model in Figure 2.13, in a Gantt-chart representation. The gripper moves too slow, so that it gets close to the handle before it is fully opened. The execution of `move_to_handle` is suspended, waiting for the gripper to have opened sufficiently. This non-nominal execution still satisfies the original plan.

2.7 Related Work

2.7.1 Mathematical Formalisms

Literature on AI and cognitive research is rich of methodologies, mathematical formalisations and languages that describe a *situation* or a chain of *events*. Those events that influence a symbolic world description may occur due to an explicit *action* execution, but an action execution is also triggered by an event occurrence. Therefore, there are multiple ways to describe a situation, depending on the *axiomatic* definitions given to the core primitives, which are **event**, **action**, **time** and **logical condition**.

Event Calculus [85, 144, 145] is a language based on logical propositions to represent events and their effects over fluents. The event is a core-primitive, and fluents capture the world status driven by events. In Event Calculus, an event is directly interpreted as an outcome of an action; in fact the terms are used interchangeably. A complementary approach is provided by the so-called Situation Calculus [100, 101], another mathematical formalisation based on first-order logic formulae. In Situation Calculus, actions are core axioms, fluents describe the state of the world, and the so-called *situation* is a finite sequence of actions. Actions require some pre-conditions to be performed, and the set of available actions and their constraints is called *domain*. Resuming, Event Calculus and Situation Calculus provide the same level of *expressivity*, but they are based on different axiomatic definitions. Analogies and differences between these methodologies have been investigated in the past [84, 15].

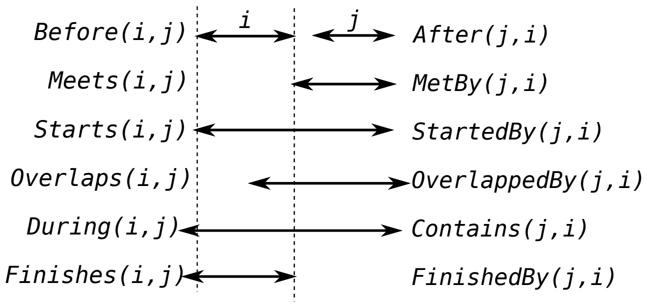


Figure 2.14: Relationships in Allen's Interval Logic [4]. Each relationship describes a time ordering constraint between a pair of finite intervals defined on the same linear time structure. Inverse relationships that describe analogous situations are on the same line. Equality relationship is not shown.

A third formalism to describe a situation is provided by the *Allen's Interval*

Logic [4, 5] (also called *Allen's Temporal Logic*). The axiomatic primitive of this approach is the *structure of time* as a linear model. Allen defines 13 primitive relationships to constrain a pair of finite interval periods, as shown in Figure 2.14. The knowledge of the temporal relations is usually represented by a hierarchical graph called *temporal network*. It is relevant to notice that Allen's Temporal Logic (and derivatives) describes *qualitative* intervals, and not *quantitative* intervals: the knowledge of the *exact* initial and final time value is not required to impose a certain relationship[10]. Intervals often represent an action, and the events are directly related to the beginning or the ending of a certain interval. Alternatives to this interpretation exist; for instance, similar relationships can be imposed between points in time (*point-time logic*) or between points and intervals (*point-interval logic*). Mathematical comparisons between temporal approaches and Event Calculus are reported in [108], and recent efforts better formulate the semantics between time and events [16]. Finally, temporal description alternatives exist; one of these is the so-called Linear Temporal Logic (LTL) [123, 11], widely known in the context of model checking and validation.

Apart of the mathematical details, there are no concrete differences in terms of expressivity provided by the above-mentioned formalisms. Depending on the use-case, one solution may provide a more concise description than another, but in practice the difference is often negligible. The main differences regard merely the axiomatic definitions, which are often matter of *"taste"*. The aim of the previous Section 2.1 and Section 1.2 was not to define yet another formalism, but to provide the set of axioms used in this work. The motivation behind the axioms choice provide in Section 2.1 is based on a *bottom-up* approach. Instead of choosing one arbitrary set of axioms, the core primitives are given by the continuous output of the *execution monitoring*, which is the *truth* information available from the system. Events are changes on these conditions, and the action attempts to modify the state of the world to a desired value. A similar approach has been taken to define a skill, that does not follow the discrete nature of the action. The *SESD* captures the execution of a skill in its dynamic and continuous evolution, exposing not only the description of the nominal behaviour, but also behaviours implemented in case of unforeseen situations. However, considering other axioms as a starting point is possible, but the final outcome would be analogous.

[10]The knowledge about the duration of the time interval is not required to describe a relationship, but it is required for scheduling with Allen's Temporal Logic.

2.7.2 Languages and Frameworks in Robotics

The previous section reported some mathematical formalisms to describe a *situation* as a constrained sequence of events or actions. Such a representation can be used for different purposes; as a narrative description of a past situation, as well as a desired (or nominal) situation to be realised. In the latter, the situation turns into a *plan*, which must be realised by one (or more) agent(s). The work presented in this chapter does not concern the plan generation, but a description of a plan having a consistent grounding in the continuous domain. The approach is based on the *Skill Dependency Graph* (*SDG*), from which a simple language is derived (*uSDL*, Section 2.3). Finally, a *SDG*-Executive mechanism has been introduced to realise a given *uSDL* description (Section 2.5).

In the robotics context, many languages and frameworks aim to solve different aspects of the planning, plan description and related run-time execution and monitoring. It follows a non-exhaustive discussion on prior works, with a special attention on the plan *languages* and *executive* features.

GOLOG and GOLEX

On the basis of Situation Calculus, the GOLOG programming language [94, 93] is probably one of the most popular between logical approaches. Multiple interpreter exist, and most of them are based on Prolog. In order to ground actions into motion primitives, GOLEX, an executive of GOLOG programs, has been introduced in [69]. GOLEX provides monitoring primitives to check the outcome of the execution of an action, which is treated as an external and atomic program; thus, no interleaving or continuous introspection is possible.

Temporal-based Frameworks

Temporal-based descriptions are widely used in Robotics. A common approach is based on Linear Temporal Logic (LTL) formalism [123, 11], for example [65, 98]. A workflow of these solutions is defined by the following steps: (i) the LTL formulae is converted to a Büchi Automaton [58], (ii) the generated automaton is directly used as a refined plan, or (iii) the Büchi Automata is transformed to a behavioural FSM or to a Behavioural Tree [45, 98, 33]. In short, the original temporal constraints expressed as LTL formula are translated to a third coordination model that can be executed. Thus, the executive layer decouples the plan constraints from a concrete solution. Reactiveness and modeling of unforeseen situations depends on the completeness of the LTL proposition, as well as on the chosen coordination model. However, it is possible to demonstrate the completeness of the generated Büchi Automaton, which is a main advantage with respect to other alternatives. As a major drawback, the LTL proposition is formulated considering an environment with no dynamic evolution apart of the nominal behaviour of the controlled agents. Therefore, it suits as an offline approach, since it is not trivial to develop an interleaved solution between the planner and the execution monitoring system.

Allen's Interval Logic are still popular nowadays; a recent example of robot task scheduling application is shown in [107]. The main limitation of this approach is due to the prior knowledge of the duration of an action. The previous assumption is acceptable to optimise a plan, but it shortly fails when the actions are dependent motion compositions; a motion execution influences the timing of another due to concurrency. In short, the role of the plan executive is underestimated.

The Stanford Research Institute Problem Solver

Another approach is given by the Stanford Research Institute Problem Solver (STRIPS) language [51], still a reference among the planning community. The original STRIPS solver is *state-space* based: similarly to Situation Calculus, the core primitive is the symbolic description of the state of the world. A STRIPS problem consists on the description of the initial and final (goal) world state. A STRIPS domain is described as a set of available actions, their effects on the world state and the necessary *pre-conditions* that guarantee a proper execution of an action. In practice, the original approach is rather limited, for instance (i) the problem domain must be *complete*, in terms of actions, agents and objects types available; (ii) *close world* assumption, that is everything must be explicitly modeled; (iii) the outcome is a *total-order* plan, with no support for concurrency. Nevertheless, STRIPS has been a pioneer result adopted by the robotic community; all the solutions that follow have been inspired by this work, at least partially. In this context, it is relevant to mention PLANEX1 [50], an early implementation of a monitor execution for STRIPS. The previous shows that interleaving planning and action execution has been a known problem in early days already, even if it has not been primary subject of investigations for several decades. The historical reason behind it is determined by the lack of interest of the AI and planning community on investing resources on such *"low-level"* issues, as well as the lack of interest from the control community, focused on providing functionalities in the continuous domain.

Procedural-based Solutions and Architectures

A renovated effort on task execution and monitor control has been delivered by the so-called *procedural approaches*. The *Reactive Action Packages* (RAP) [53, 54] are designed to bridge the gap between the symbolic action to a low-level skill, defined in the continuous, real-time domain. A RAP Executor reacts to primitive events by activating (or deactivating) a certain skill. The skills are stored and connected each other through a so-called *skill network*, exhibiting a dependency relationship (not formally expressed within the RAP framework). The semantic of the outcome obtained by a skill execution is rather limited to a *failure* or *success* status. In fact, the skills are mostly commands or processes that implement a control law, while reactions are determined by defined *memory-rules* triggered by task completion or external signals. Furthermore, the *memory-rules* mechanism in RAP offers the possibility to interpret an event given a certain context. RAP framework also supports hierarchy through a tree structure composition of skills.

In the same vein, the *Task Control Architecture* (TCA) [147] and its extension, the *Task Description Language* (TDL) [148], is an alternative to describe Robot Tasks. The TCA/TDL allows concurrent planning and execution through hierarchical task decomposition. In addition, the framework provides explicit constructs to define temporal constraints that allow synchronization between tasks. Other alternatives exists, among which the *Procedural Reasoning System* (PRS) [72, 92] or an alternative *TaskNets* framework presented in [161].

The above-mentioned procedural solutions do not strongly differ between each other. Furthermore, it is worth to notice that these frameworks are results of investigations conducted in the same period of time; none of them was meant to improve the state-of-the-art of the others. The main difference regards the methods for distributing the required functionalities on the overall architecture, which is often a technical matter rather than a modeling issue. In fact, most of these approaches conform to the so-called *Three-Tiered Architecture* (3T-A) [24, 83] shown in Figure 2.15. These task executives, which resides in the second layer of the 3T-A, have been designed to be integrated with legacy planners or control middleware. Thus, one approach better adapts against legacy functionalities than another. For instance, TDL implementation suits to C/C++ applications, where concurrent skills are control law instances deployed in *OS threads*. Another example is the recent *Light-Weight Robot Coding for Skills* (*LightRocks*) DSL [162]. *LightRocks* extends the *TaskNets* approach [161, 52], which is based on manipulation primitives, to a hierarchical skill description based on UML/P statecharts.

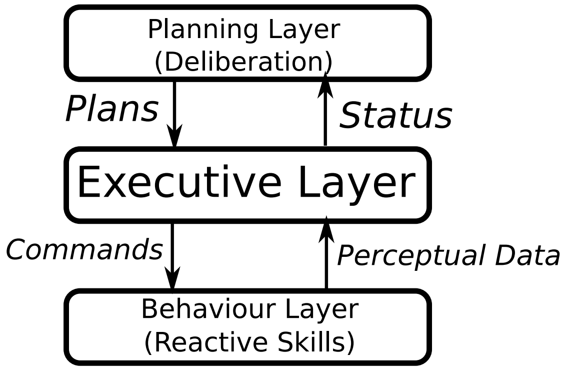


Figure 2.15: A Three-Tiered Control Architecture [24, 83] adopted in multiple frameworks, such as the Task Control Architecture (TCA) [147, 148] and the LAAS Architecture [3, 125]. The naming convention on each layer may differ from framework to framework. For instance, planning, executive and behaviour layers have been called *decisional*, *execution control* and *functional* levels in the LAAS Architecture. However, they all conform to the same architectural description.

The Plan Execution Interchange Language

An *Automata-based* approach has been follow to design the *Plan Execution Interchange Language* (PLEXIL) [170]. PLEXIL has been developed by the Jet Propulsion Laboratory (JPL) at NASA, in order to replace the task executive on the *Coupled Layer Architecture for Robotic Autonomy* (CLARAty) [112], previously based on the TDL. PLEXIL has been successfully applied to various spacecraft applications, such as the *K10 Rover*, the *Drilling Automation for Mars Exploration* (DAME) [63], and other automated operations on the *International Space Station*[11] (ISS).

Similar to the TDL, a *node* in PLEXIL represents an action, which is linked to an external command to execute. The node status reflects the run-time information on the command execution; an enumerative list of possible status follows: *waiting*, *executing*, *finishing*, *failing*, *finished*. Also, the node provides an outcome property that must match with one of the following status: *success*, *failure*, *skipped*. Nodes follow a hierarchical tree structure, thus a node can have a list children nodes, but each child has only one parent. The execution of the node is driven by a set of constraints over boolean expressions. PLEXIL differentiates between the so-called *gate conditions* and *check conditions*. The former are monitored continuously; the latter are evaluated once, in a well-

[11] `https://www.nasa.gov/mission_pages/station/main/index.html`

known moment of the node life cycle. For example, the *EndCondition* is a gate condition and, if that holds, a *PostCondition* is evaluated to ensure whether the desired outcome has been achieved by the action. In the same vein, the *StartCondition* is a gate condition that evaluates if the node can be executed and, if that holds, an additional *PreCondition* acts as a *guard* mechanism to prevent the node activation. If no activation constraints are imposed, the node is immediately executed. PLEXIL *explicitly* provides an *InvariantCondition* that must be valid along the overall execution of the action. Furthermore, PLEXIL provides constructs such as *RepeatUntilCondition* to invoke the same command multiple times.

The Cognitive Robot Abstract Machine

An alternative architecture is provided by the Cognitive Robot Abstract Machine (CRAM) [14]. CRAM is a framework that well integrates a perception pipeline and a knowledge-space database [158, 159] (the so-called *KnowRob*). CRAM includes a Cram Plan Language (CPL) that allows to describe a plan as a partial-order sequence of actions. CPL is implemented in Lisp programming language, and it supports description of both sequences and concurrencies of actions, with guard predicates on an action failure. The level of expressivity of CPL is not dissimilar to a Behavioural Tree approach [45]. The advantages of the CRAM system are related to a reactive CRAM-Executive [173] that, in case of an action failures, triggers a Prolog-based reasoner which extends or modifies the original plan to a feasible solution. This plan enrichment is obtained by inferencing the knowledge space, which has been updated at run-time. However, CRAM does not offer any particular feature about managing the action execution. The actions available are programmed motion primitives, executed *"as-they-are"* by a motion server. Thus, an action can be activated (or deactivated), with few parametrization options, and the interface with the motion stack is implemented in a client-server fashion. Once again, the latter is determined by the choice of the middleware that integrates the motion control, that is the Robot Operating System (ROS) [126].

The Planning Domain Definition Language

It is not possible to conclude this section without mentioning the Planning Domain Definition Language (PDDL) [56, 59], a recent and evolving effort of the planning community. The PDDL provides a common DSL to describe a planning problem and its domain in a generic fashion, decoupling from the symbolic solver adopted. The development of PDDL is driven by the necessity of the planning community to formulate a set of problems to benchmark existing solvers. PDDL can specify problems in STRIPS or other formalisms, including temporal planning, scheduling, probabilistic problems and so on. PDDL does not offer any facility regarding an execution layer. So far, the Robotic Community made some efforts to bridge the gap between classical discrete planners and continuous motion planning [156, 109]: these solutions reside in the planning, and in case of environmental changes, replanning is preferred to active control.

2.8 Conclusions

This chapter presents a formal task executive framework based on a *hierarchical hypergraph* structure, the so-called *Skill Dependency Graph*. The *SDG* fully describes a nominal plan as a composition of skills and their dependencies, which constrain the skills between each other. Each skill is modelled as a *hybrid system*, and this chapter has focused on describing the set of discrete behaviour that a skill exhibits. A *SDG Executive* is introduced as a software entity that activates a given *SDG* model. Furthermore, a *micro Skill Dependency Language* is proposed as a specification language to compose a *SDG* model. In this way, *SDG* models are *partially-ordered* plans, **reacting to non-nominal situations**. A use-case is discussed in Section 2.6.

The introduced mathematical foundation is equivalent in expressivity with respect to other formalisms, but the adopted axioms, derived by a *bottom-up* approach, directly bind the logical primitives to the continuous domain. The latter has several practical implications, especially in the context of specification of robotic motions. Among those, the most relevant is the capability of specify a **conditional concurrency** execution between skills. In fact, all the existing task executives offer the capability to model *sequential* and *concurrent* skill executions, but none of those allow to specify an arbitrary guard condition along the continuous behaviour of a skill, imposing an execution serialization only when needed. This feature **improves the expressivity of a plan specification**, allowing **online adaptation** by skill scheduling.

To the best of author's knowledge, only PLEXIL [170] includes the concept of logical `invariant` constraint. However, a PLEXIL action is implemented as an atomic command, which can be only activated or not. Thus, no anchoring is possible at the continuous level; instead, the skill model adopted in the *SDG* formalisation allows to **bridge the `invariant` constraint between discrete and continuous domain**, at the cost of an extra modelling effort. The latter became *essential* if the skill is implemented as a set of constraints, formulating a COP that generates the motion in the continuous domain. The following chapter will cover the above-mentioned topic.

Chapter 3

Constraint-based Task Specification based on Expressions between Geometric Primitives[1]

Constraint-based programming is a well known approach to specify and generate robotic motions. A primary feature of this approach is given by the *composability* of the task specification: a task can be defined as a composition of other constraint-based tasks, as well as the objective function that drive the generated behaviour. Hence, satisfying multiple constraints corresponds to execute multiple sub-tasks *concurrently*. This chapter presents a formal *Domain Specific Language* (DSL) to specify constraint-based tasks. Instead of focusing on a specific feature provided by an underlying numerical solver (Section 3.1), this language allows to define constraints over geometric expressions, that are relationships between geometric primitives (Section 3.2). In addition, the language includes a way to express the control behaviour imposed for the constraints resolution, compatible with most of the existing solvers. This work aims to provide *i)* a formal model to decouple the task specification from the underlying numerical solver, *ii)* a geometric-based approach, which is more intuitive than the alternatives available, but also *iii)* it carries semantic information, enabling high level reasoning, such as automatic generation of the

[1]The content of this chapter is partially based on the research presented in [26], which has been carried out with the contributions of other co-authors.

constraints, as well as executable feasibility over the robot capabilities. Finally, task specification examples will be provided and validated with an experimental setup (Section 3.3).

3.1 Introduction and Related Work

Constrained Optimization is a well known mathematical process that computes a *solution* $\mathbf{x}$ (if any exists) that minimises the value of a given *objective function* $f(\mathbf{x})$. Such a solution must lie in the so-called *feasible space*, defined by a set of constraints expressed as *equality* or *inequality* functions. A generic formulation of a *Constraint Optimization Problem* (COP) can be written as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}), \\ \text{subject to} \quad & g(\mathbf{x}) = \mathbf{c}, \quad \text{Equality Constraints}, \\ & h(\mathbf{x}) \leq \mathbf{d} \quad \text{Inequality Constraints}. \end{aligned} \tag{3.1}$$

As a side note, a COP must not be confused with a *Constraint Satisfaction Problem* (CSP). The latter regards only the satisfaction of the constraints, often expressed as *integer* constraints. CSPs are widely used for planning in the symbolic domain.

COP formulations are widely used in many robotics contexts, among with *motion planning* and *motion control*. The former focuses on generating trajectories that complies to kinodynamic constraints, minimising path length, time, and energy required for the execution or a combination of these; examples are [171, 38, 40]. The latter focuses on generating an instantaneous *control action*, expressed as desired velocities, accelerations or forces on the robot joints (or a combination of those, see [142]). The objective function defines the general behaviour, often formulated to optimise energy consumption, in one form or another (*e.g.*, minimal velocities, accelerations or forces applied to the joints); constraints can be defined in both joint or *task* space [135]. In this context, the term *task specification* often refers to a specific formulation of a COP to perform a given task.

Constraint programming enables a set of advantages over classical motion specifications, among which:

- ***composability* of the constraints:** most of the constraints does not involve the whole platform, hence satisfying multiple constraints is possible; combining constraints is a way to generate a task by composition of multiple sub-tasks, which are performed concurrently;

- ***redundancy resolution*:** a COP formulation allows to exploit and optimise the behaviour on the overall *degrees of freedom* (DOFs) provided by the robotic platform;
- ***portability* of the task specification:** under certain conditions, the same task specification can be executed by another robot platform; these conditions often depend on the *robot capabilities*, *i.e.*, a specification can be incompatible whereas the robot does not provide certain features or tools;
- ***uncertainties:*** some frameworks (*e.g.*, [36]) provide a systematic approach to model and estimate geometric uncertainties online, such that the control action is adapted accordingly;
- ***reactiveness*:** in the instantaneous version, the control action is computed at a frequency rate compatible with the real-time requirements of the application.

However, formulating a COP is not always trivial. To this end, several frameworks provide a specific toolchain, often addressed to a specific *solver* implementation. Some examples are: *i)* the *Stanford Whole-Body Control* framework (SWBC) [141] that deals with force-resolved schemes, *ii)* the *instantaneous Task Specification using Constraints* (iTaSC) [36] introduces feature and uncertainties variables, allowing to define tasks in complex task spaces and to treat systematically geometric uncertainties. In the iTaSC, the optimisation problem is defined as a least-squares problem, which uses the Moore-Penrose pseudo-inverse algorithm [116]; a further extension supports both equality and inequality constraints [41]. Another family of solvers is provided by the *iii) Stack of Tasks* (SoT) [96, 97], originally based on a Hierarchical Quadratic velocity-resolved scheme [48], then extended to force-resolved scheme [133, 42]. A velocity-based scheme based on a *Sequential Quadratic Programming* (SQP) solver (qpOASES, see [49]) is adopted by the *iv) openSoT* [129], as well as the *v) expression-based Task Specification Language* (eTaSL) [2]. The latter differs from the others due to the introduction of *expressiontrees* for automatic differentiation purposes (*e.g.*, Jacobian computation). Alternative formulations on the same SQP approach can be found in [9]. Finally, *vi)* hybrid kinematic-dynamic scheme have been investigated in [142, 143]. This non-exhaustive overview shows the increasing popularity over the COP methodology, and, functionality-wise, demonstrates its maturity. Furthermore, none of the above-mentioned framework is meant as a replacement of the others, but as an alternative which better adapts to a concrete context.

However, none of the above-mentioned frameworks provide a practical solution to compose a task specification: there is no common methodology among the frameworks, many tuning parameters are required, and the learning curve is quite steep for a non-expert. As a matter of fact, experts stick on one solver or methodology, without adopting any alternative; non-experts are reluctant to use these methods, opting for motion planning solutions.

To the best of the author's knowledge, only few initiatives have been taken to formally describe a task-centric application. Starting from the seminal work of Mason [99], a more formal language to describe compliant motions is given by the *Task Frame Formalism* (TFF) [30, 82], later revisited in [86]. However, these approaches does not scale to more complex robots and tasks. Furthermore, it is often implicitly assumed that a robot is capable of performing a certain motion, or it provides the required capabilities for the motion execution. Examples are guarded motions over force-based constraints: not all robots provide a force sensing capability. Another work is the iTaSC-DSL [168] that supports only a task specification based on the iTaSC methodology. The recent eTaSL framework [2] provides another example of task specification strictly related to the underlying solver capabilities.

This chapter introduces an alternative DSL for constraint-based task specifications, which is not only independent from the numerical solver or methodology adopted, but it is also translated and implemented in existing frameworks. To this end, the task specification formalises geometric primitives as first-class citizens, which are both feature of concrete items instances and target of relationships that describe a desired world transformation.

Another drawback of a COP formulation can arise when multiple sub-tasks are combined together, generating a situation of *conflicting constraints*. For instance, a conflict between constraints occurs when *i)* some inequality constraints are partitioning the same space in disjoint sets, *ii)* multiple equality constraints defined in the same space are imposed, and when *iii)* a robot is *over-constrained* [132, 165], thus there is no redundancy available to fully satisfy the constraints. Luckily, all of the above-mentioned frameworks provide a mechanism to deal with *conflicting constraints*, already in the continuous domain. The most common solutions are *i)* constraint weighting between conflicting constraints [135, 44, 97, 2]) and *ii)* constraint prioritization [110, 140, 36, 43]. However, the implementation of such mechanisms is solver-dependent; as an example, the iTaSC [36] implements priorities through null space projection [10], while the eTaSL [2] implements priorities by introducing *slack variables* in the QP formulation. Lastly, solutions in the discrete domain are possible, such as reformulations of the objective function (*e.g.*, removal of a term) or the removal of a conflicting constraint, as adopted by the SoT [96].

Resuming, the aim of the chapter is not to introduce a novel COP solver or methodology. Instead, the purpose is to design a task specification that *unifies*

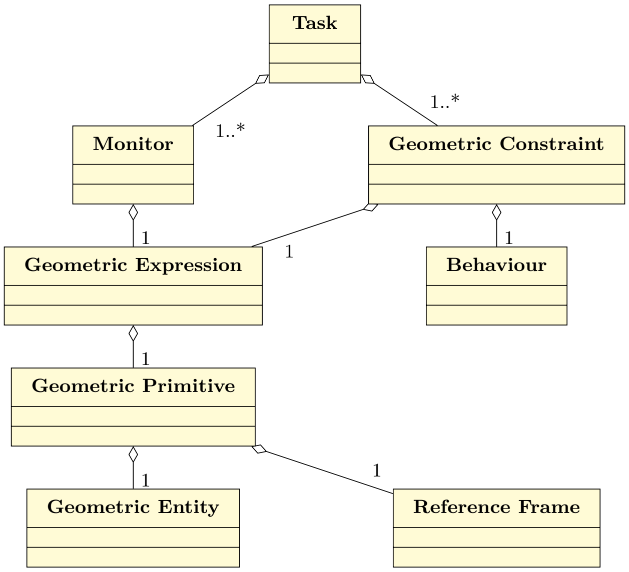


Figure 3.1: Overview of the task specification objects in a UML class diagram style. This Figure is illustrative purposes only; several elements, such as attributes of the classes, are omitted.

the existing frameworks.

3.2 Geometric-based Task Specification DSL

This section introduces a motion task specification based on geometric constraint expressions. This DSL aims to abstract *i)* the space where tasks are expressed, which is defined in terms of symbolic expressions, *ii)* the type of control to be enforced on such a space (the behaviour), and *iii)* the monitors required to detect an achievement over the desired behaviour. One of the design drivers of the language is the composability (by aggregation) of the language elements, as it is shown in Figure 3.1. An implementation of the language is grounded in JSON-schema, which is reported in appendix A.2. Therefore, all the examples in Section 3.4 *conforms to* that meta-model.

3.2.1 Geometric Expressions between Geometric Primitives

This section introduces the geometric concepts of *i) geometric entities* (a geometrical feature such as a point, a versor, a line, a plane and so on), *ii) geometric primitives* (entities expressed in a frame), and *iii) geometric expressions* (a scalar function that relates primitive). Expressions can be non-geometric as well (*e.g.*, joint expressions).

Expressions are a relationship that maps *joint-space* values and measurements, to *task-space* values (controlled, measured, and so on). The use of expressions is two-fold: *i)* to compute "deviations" from (task-space) positions, or *ii)* to compute the "Jacobian" (the partial derivative of the expression w.r.t. joint angles) that relates joint velocities (forces) to generalized velocities (torques).

Since our focus is *task specification*, no assumption is made on how values and Jacobian are computed (*e.g.*, analytically, numerically, solving implicit equations, and so on).

Geometric Entities

The elementary *geometric entities* considered are the *point* and the *versor*; both have *three parameters* that can be represented by triples (x, y, z). Additionally, the versor must comply with unitary norm constraint. Combinations of points and versors yield to other two well-known geometric entities, *line* and *plane*, each with *five free parameters* in its representation. This representation is *non-minimal*, since it is known that a minimal representation of a line requires only four parameters. Nevertheless, the chosen representation is intuitive, obtained by composition, and it simplifies the specification. Furthermore, it carries an extra information, which is an orientation with respect to the line origin. The latter is very convenient; for instance it allows to specify *projection relationships* described below.

Geometric Primitives

Entities need a *frame* to ground their *coordinate representations*: so, a *geometric primitive* associates a *geometric entity* with a *frame*. The four possible geometric primitives are reported in Table 3.1, along with the associated mathematical symbol. Formal *meta-models* of *point entity* and *primitive* are given in Listings 3.1–3.2. As a reminder, a meta-model represents all the *formal constraints* that every individual instance must conform to. In turn, the chosen *meta-meta-model* to describe this work is based on JSON-Schema.

```
{ "id": "http://.../point#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Point Entity",
  "type": "object",
  "properties": {
    "x": {
      "type": "number",
      "description": "coordinate along x-axis"},
    "y": {
      "type": "number",
      "description": "coordinate along y-axis"},
    "z": {
      "type": "number",
      "description": "coordinate along z-axis"},
    "type": { "enum": ["point"] }
  },
  "required": [ "x", "y", "z", "type" ],
  "additionalProperties": false
}
```

Listing 3.1: Formal JSON meta-model of a *point entity*.

```
{ "id": "http://.../primitive#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Geometric Primitive",
  "type": "object",
  "properties": {
    "object_frame": {
      "type": "string",
      "description": "reference frame"},
    "entity": { "$ref": "#/definitions/entity" }
  },
  "required": ["object_frame", "entity"],
  "additionalProperties": false,
  "definitions": {
    "entity": {
        "oneOf": [
          { "$ref": "http://.../point#" },
          { "$ref": "http://.../versor#" },
          { "$ref": "http://.../plane#" },
          { "$ref": "http://.../line#" } ]
} } }
```

Listing 3.2: Formal JSON meta-model of a *geometric primitive*.

Geometric Primitive	Symbol	entity composed of
point expr. in $\{w\}$	$\boldsymbol{p}_{\{w\}}$	scalars x, y, z
versor expr. in $\{w\}$	$\widehat{\boldsymbol{n}}_{\{w\}}$	scalars x, y, z s.t. $\|\| \cdot \|\| = 1$
line expr. in $\{w\}$	$\overline{\boldsymbol{n}}_{\{w\}}$	point *origin*, versor *direction*
plane expr. in $\{w\}$	$\mathfrak{P}_{\{w\}}$	point *origin*, versor *normal*

Table 3.1: Summary of geometric primitives, grounded in frame $\{w\}$.

	point	line
point	point-point distance	
line	line-point distance	distance btw lines
	projection of point on line	projection (p1-f1)
		projection (p2-f2)
plane	point-plane distance	

(a) Geometric expressions on distances.

	versor	plane
versor	angle btw versors	
plane	incident angle	angle btw planes

(b) Geometric expressions on angles.

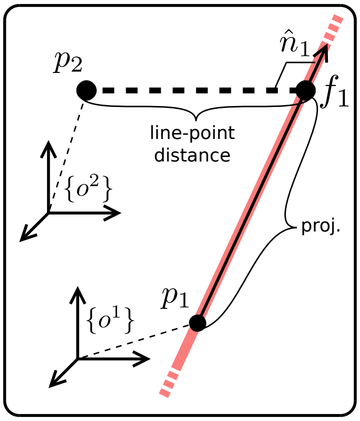
Table 3.2: Summary of geometric expressions between primitives.

Geometric Expressions

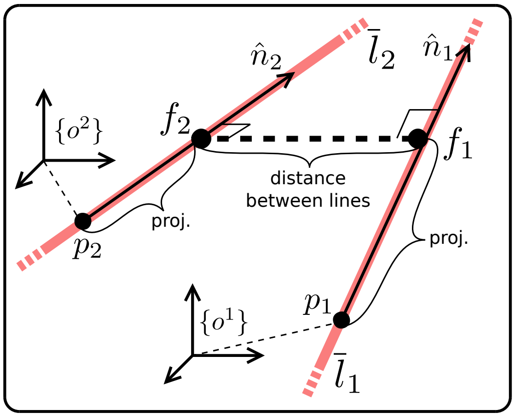
A *geometric expression* is a relationship between a pair of well-defined geometric primitives. The list of primitives in Table 3.1 is not exhaustive, but suffices for most scalar expressions that describe positioning between pairs of objects. For example, Table 3.2 gives *distance* and *angle* expressions. Most of the table entries are self-explaining; nevertheless the expressions *line-point* and *line-line distance* require an additional explanation to avoid any misunderstanding:

- The *line-point* entry (Figure 3.2a) has two expressions: the *line-point distance*, that is the distance between the point $\boldsymbol{p}_2$ and the point of shortest distance $\boldsymbol{f}_1$, and the *projection of point on line*, that is the distance between the points $\boldsymbol{p}_1$ and $\boldsymbol{f}_1$.
- The *line-line* entry (Figure 3.2b) has three distance expressions: the *distance from lines*, distance d between the two lines, the distance between the point $\boldsymbol{p}_1$ and the minimum distance point $\boldsymbol{f}_1$, and the distance between the origin of the line $\boldsymbol{p}_1$ and the minimum distance point $\boldsymbol{f}_2$.

A relevant remark is that all the above-mentioned distances are *signed*; the latter is a useful property that increases the expressivity of the proposed DSL.



(a) Line 1 is expressed in $\left\{o^1\right\}$, Point 2 in $\left\{o^2\right\}$.



(b) Lines 1 and 2 are expressed in $\left\{o^1\right\}$ and $\left\{o^2\right\}$, respectively.

Figure 3.2: Graphical representations of the five possible relations between a point and a line (3.2a), and between two lines (3.2b).

Expressions in Joint Space

Geometric expressions are used to describe constraints on the task space. However, many cases need to express constraints in a robot's joint space, the most obvious cases being limits in torque, position, or velocity. Joint space expressions, as well the constraints defined upon these, are platform dependent and they change from robot to robot, thus they are not task specific.

Composite Expressions

Many constraint specifications require complex expressions, either as a *combination* of the elementary expressions above, or based on more complex geometrical *shapes*, or specified directly as geometric "curves". The adopted approach is *to enumerate* all models that one requires in a specific task context, since with this approach it is trivial to extend the enumeration list. The following are few examples of such higher complexity expressions:

- **Non-scalar expressions** or multi-dimensional expressions, as rotations in space. Rotations are often used when full orientation is constrained. However, a analogous results can be achieved by considering a set of *angle between versor* expresssions.

- **Sensor-space expressions**; in some applications, constraints are directly expressed in a sensor space, *e.g.*in visual servoing with eye-in-hand camera. In cases where such a relation cannot be expressed by means of geometric expression, a new expression must be formalised, and the underling implementation realised:

  $$y = f(\boldsymbol{\chi_u}, \boldsymbol{q}, \dots),$$

  where $\boldsymbol{\chi_u}$ is the relative position of the measured object with respect to the camera, y is the measured output, and $\boldsymbol{q}$ refers to the robot configuration.

3.2.2 The Behaviour

Geometric expressions are a useful tool to specify constraints in the "output space", but *how* should these constraints be satisfied has not been discussed yet. The latter refers to the concept of *behaviour* specification, which is a way to describe *how* to achieve a certain result, delegating the concrete implementation of the control law to the underling solver.

An enumerative list of behaviours covered in this chapter are:

- **Positioning**, that describes position regulation problems;
- **Move**, to specify direction and rate of motion, rather than only the desired final position;
- **Physical Interaction**, to control force or impedance occurring in physical contact;
- **Compliant positioning**, to control the position of the system, while allowing for physical compliance in order to cope with unexpected or partially modelled contacts;
- **Limits**, to reduce the space of feasible solutions, preventing the robot from going in undesired positions or to exert excessive forces.

Table 3.3 matches behaviours with the needed characteristics of the system, in terms of: *i)* controller type, *ii)* type of set-point that must be specified, such as position, velocity and forces, either as a constant, provided by a trajectory generator or another external source, and *iii)* type of constraint, *i.e.*, equality or inequality.

Obviously, these functionalities should be provided by the system developer, and they should be achievable by the robot that performs the task. Thus, not all the behaviours are available on a controlled system. The latter re-call to the concept of *robot capabilities*: a task specification can be executed only if the robot offers all the required features to guarantee a correct realisation, among with the relative control algorithm (software) and sensing capabilities (*e.g.*, force sensor).

Referring to Table 3.3, the features of the implemented control algorithms should comply to the follow requirements: *i)* in the first three cases, the goal is to have a zero steady state error in either position, velocity, or force, *ii)* in the compliance mode the control algorithm regulates the position, but allowing deviations proportional to force, and *iii)* with limits, only the bound satisfaction is sufficient, and no motions are imposed if the latter holds. In addition, the first four behaviours need a parameter to indicate *i)* the time constant of the error, in the first three cases, or *ii)* the relation between angular or linear displacement (along the output direction) with respect to the disturbance (force or torques respectively).

With the first three control laws, we seek an asymptotically zero converging error in position, velocity, or force, respectively, and the error dynamic should be "analogous" to a first order system. In the case of position control:

$$\dot{y}_d^\circ = \boldsymbol{K}_p(y_d - y), \tag{3.2}$$

where $\dot{y}_d^\circ$ is the velocity actuated (see [36]), and the gain $\boldsymbol{K}_p$ is the *specification*, and whose dimension is [$^1/_s$] regardless of the constrained variable; a feedforward term ($\dot{y}_d$ or $\dot{\lambda}_d$, in Table 3.3) could also be taken into account.

The compliant motion behaviour, instead, achieves a given displacement from a rest position as response to disturbances (*e.g.*, an external force). Henceforth, the tuning parameter represents a desired apparent stiffness:

$$\delta f = \mathbf{K}\delta y. \tag{3.3}$$

Lastly, in case of limiting-type behaviours, parameters are considered optional, since in many cases (*e.g.*, joint position or effort limits), the desired interval should not be violated (and no convergence rule toward limits is needed). On the contrary, in cases where limiting behaviours are used to define "soft constraints" (*e.g.*, auxiliary constraints as described in the following section), these gains can be used to assess a smooth convergence toward the limits following the control equation 3.3.

The described behaviours cover most of the tasks proposed in literature, but it is not exhaustive. If the need arises, it is possible to extend the enumerated list with additional behaviours (or add additional parameters to the proposed ones), provided that controllers exist that are able to execute them.

3.2.3 Constraints, Monitors and Task

Constraints

A constraint is defined as a composition of two primitives, that is *i)* an expression, and *ii)* a set-point specification, as a constant value, or as a generated trajectory or any other external source. Furthermore, a constraint must be linked to a valid *behaviour* in order to be executed.

In all but the simplest tasks, several constraints are imposed together. Since the initial conditions, the environment, and other aspects can be unknown at the time of defining the application, or vary between executions, a task could result in conflicting objectives that cannot be achieved simultaneously. For this reason, a well-formed specification should express explicitly in which way conflicts should be handled at run-time. Therefore, a semantic tag is attached to each constraint; such a tag is interpreted at run-time, and in case of conflicts a certain policy is applied. The number of tags is an arbitrary choice that depends on the level of control granularity required by the application. In most of the applications (see [137, 136]) the following three levels suffice:

- **Safety constraints**: constraints that are necessary for the *robot platform* or critical surroundings integrity, such as hardware limitations, non-desired collisions, sustained balance in humanoid robots, and so on. These constraints are usually formulated as inequalities, thus reducing the space of feasible solutions in which lower level constraints can be fulfilled. Constraints executed in this level should not be conflicting.
- **Primary constraints**: those constraints that, if satisfied, trigger a logical step-forward along the nominal plan execution,
- **Auxiliary constraints**: also called *soft-constraints*, are those that facilitate the execution of primary constraints, but they can be violated without causing a failure over the desired behaviour; an example is the optimization of the robot pose configuration with respect to the manipulability index, [25], others are gazing, elbow configuration and so on.

Behaviour	**needs**:									**Specification**
	controller	setpoints (one of)					Force measurement	constraints		
		y_d	$\dot{y}_d$	y_d, $\dot{y}_d$	λ_d	λ_d, $\dot{\lambda}_d$		$=$	$<$	
Positioning	Position	✓		✓				✓		dominant pole [1/s]
Move	Velocity		✓					✓		dominant pole [1/s]
Physical interaction	Force				✓	✓	✓	✓		dominant pole [1/s]
Compliant motion	Impedance	✓					✓	✓		Stiffness [N/m] or [Nm/rad]
Position Limit	Position	$\checkmark^{\times 2}$							✓	(dominant pole [1/s])
Velocity Limit	Velocity		$\checkmark^{\times 2}$						✓	*(dominant pole* [1/s]*)*
Force limit	Force				$\checkmark^{\times 2}$		✓		✓	*(dominant pole* [1/s]*)*

Table 3.3: List of behaviours: each behaviour is related to the type of control and its specification, the type of set-point, the needed measurements (position measurement is always needed), and the related constraint (either equality or inequality). Inequalities needs two set-points, representing the upper and lower bound values. Specification is optional for limiting behaviours.

A conflicting situation is solved by a pre-defined policy on the indication of the semantic tags associated to the constraints involved. Recalling from Section 3.1, conflicting constraints are solved by implementation-dependent methods, among with: *i)* weighting of a conflicting term in the objective function, *ii)* prioritization of constraints, achieved mostly with null space projectors, and *iii)* removal of the conflicting constraint. Since the purpose of this chapter is to present a task specification, no further details are provided in this context.

Monitors

The importance of the *execution monitoring* has been discussed already in Section 2.1. The proposed DSL introduces a way to configure a *monitor function*, promoting the usage of the geometric expressions. In details, a monitor can either observe:

1. a controllable variable (for example a variable that is used as constraint), or

2. a measured quantity that is influenced by the robot actions, but cannot be directly controlled. This quantity can be measured in terms of: *i)* a variable that lives in the space described by an expression, or *ii)* an external monitored value.

For the cases 1) and 2*i*, a monitor is defined as: *i)* an expression (that specifies the space where the variable lives), *ii)* a condition name (*e.g.*`finished, failed`), *iii)* the (monitored) variable type (`POSITION`, `VELOCITY`, `FORCE`), that is expressed in the space defined by the expression, *iv)* a comparison type ($<$, $>$, $\in$, $\notin$), *v)* reference value(s). The external monitor (case 2*ii*) allows for composability with external sources, for example, time-out or safety events.

Task

Finally, the formal definition of a Task wraps the previous items together; it includes *i) at least one* primary constraint, and *ii) at least* one (end-of-task) monitor. *Optionally*, it can have *iii)* safety constraints, and *iv)* auxiliary constraints.

Thus, the minimum specification of the task corresponds to a constraint (an expression, a behaviour that rules which kind of control must be used, and a reference value) and one monitor (that dictates the end of the task execution).

Having a well defined end-of-task monitor is instrumental to define the post-condition(s) of the task to be used in plan reasoning and scheduling algorithm (see Section 2.1).

3.3 Task Specification Examples

This section shows two different task specifications related to different case studies, namely an *open a drawer* (previously discussed in Chapter 2) and a *spanning a surface* scenarios. The task specification is transformed to a COP, which is solved by a numerical algorithm. The presented results have been obtained in both simulation environment and on a real robotic system. The execution is validated by the behaviour performed, thus by monitoring the satisfaction of the imposed geometric constraints. However, this chapter focuses on the *"feedforward"* part of the whole control problem, which is fully realised on a model, whether or not adapted at run-time. It is beyond the scope of this chapter to discuss about the quality of the generated motion, or on the control methodology adopted, since there are no contribution in that perspective. The main aim of this section is to propose a possible realisation by means of concrete examples. For the sake of clarity, the discrete coordination of the robotic tasks does not adopt the *SDG* methodology discussed in Chapter 2, which it will be illustrated in Chapter 4. Instead, the adopted coordination model is based on a FSM, mostly driving the switch of a strict sequence of tasks encoded *a priori*.

3.3.1 Numerical Solver

As mentioned in Section 3.1, the focus of this chapter is neither on the numerical solver or on the methodology adopted to ground the proposed DSL. However, in order to provide an executable task specification, the choice over a numerical solver is required. This section briefly illustrates the velocity-resolved solver employed to perform the motion controller, which is a modified version of the solver provided in eTaSL framework [2].

Let $\mathbf{x}$ be optimisation variable written as $\mathbf{x} = \begin{bmatrix}\dot{\mathbf{q}}^T & \varepsilon^T\end{bmatrix}^T$, where $\dot{\mathbf{q}}$ is the control action (*i.e.*, n_r joint velocities of the controlled robot), and ε is vector of n_s slack variables that implements *primary* and *auxiliary* constraints in the solver (further details are provided below). The instantaneous control action is computed by solving a SQP written as:

$$\underset{\mathbf{x}}{\text{minimize}} \qquad \mathbf{x}^T\mathbf{H}\mathbf{x}, \tag{3.4}$$

$$\text{subject to} \qquad \mathbf{L}_A \leq \mathbf{A}\mathbf{x} \leq \mathbf{U}_A, \tag{3.5}$$

$$\mathbf{L} \leq \mathbf{x} \leq \mathbf{U}, \tag{3.6}$$

where Eq. 3.6 expresses the velocity limits that bound the feasible space; hence, this equation is filled in by the *safety constraints* described in the task specification. The behaviour of the remaining geometric constraints is grounded by Eq. 3.5. In details, each geometric constraint[2] i defines a *task function* $e_i(\mathbf{q}, t)$ as suggested in [135]. The desired evolution of such a task function is imposed by a first order system with time constant K^{-1} (*i.e.*, a simple proportional controller); for equality constraint,

$$\frac{d}{dt}e_i(\mathbf{q}, t) = -\mathbf{K}e_i(\mathbf{q}, t), \tag{3.7}$$

which slightly differs in case of inequality constraint,

$$\frac{d}{dt}e_i(\mathbf{q}, t) \leq -\mathbf{K}e_i(\mathbf{q}, t). \tag{3.8}$$

The evolution of the task function can be re-written as partial derivatives,

$$\frac{de_i}{dt} = \frac{\partial e_i}{\partial t} + \sum_{j=1}^{n_r} \underbrace{\frac{\partial e_i}{\partial q_j}}_{\mathbf{J}_i} \dot{q}_j, \tag{3.9}$$

and combined with Eq. 3.7 (or Eq. 3.8),

$$\mathbf{J}_i\dot{\mathbf{q}} = -\mathbf{K}e_i - \frac{\partial}{\partial t}e_i + \varepsilon_i. \tag{3.10}$$

In the latter, ε_i (*e.g.*, slack variable of the constraint i) is added to implement a weighting policy without causing infeasibility of the QP problem (the feasible space is determined only by the *safety constraints*). As a remark, the above formulation is also adopted by the SoT [97]. To resume, this solution imposes an exponential decay to the behaviour of the constraint to be satisfied; a possible drawback is the introduction of discontinuities in the computed task space, in particular when a constraint is inserted. To avoid such situations, a velocity profile can be attached to the controller as *feedforward* term, imposing a desired

[2]All the geometric constraints represented in the DSL are *scalars*, therefore no special care is required.

and customizable behaviour, still defined in the task space. An alternative solution is to adopt an adaptive gain function instead of the time constant K^{-1}.

To conclude, the resulting Hessian matrix that defines the objective function is:

$$\mathbf{H} = \begin{bmatrix} \mu \mathbf{W}_r & 0 \\ 0 & \mu \mathbf{I} + \mathbf{W}_s \end{bmatrix}, \tag{3.11}$$

where *i)* μ is an arbitrary value large enough to guarantee that the Hessian matrix is positive definite, *ii)* $\mathbf{W}_r$ are weights on the joint space, and *iii)* $\mathbf{W}_s$ are weights on the *primary* and *auxiliary* constraints. In the latter, an arbitrary weight-based policy must be defined to differentiate between *primary* and *auxiliary* constraints. To solve the QP problem (Eq. 3.4), the qpOASES solver [49] is adopted.

As a remark, this solution is not the only implementation available; in fact, within the European Project FP7-*RoboHow.cog* [128], the proposed Geometric-based DSL (Section 3.2) has been also grounded in the SoT [96] framework[3].

3.3.2 Open a Drawer Scenario

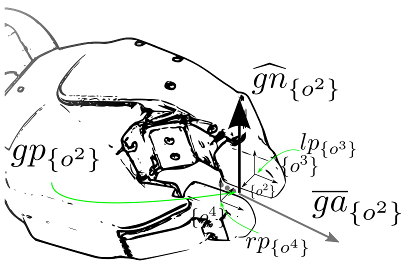
A first case of study is the already discussed *"open a drawer scenario"*: Section 2.6 briefly illustrates the constraint-based motions in a narrative way, instead this section finalises the description of the task formally. Furthermore, both simulation and real experiments are reported below; that allows to show slightly different task specifications for the same application, enforcing the concept that from a symbolic action multiple grounding are possible.

Simulation Validation

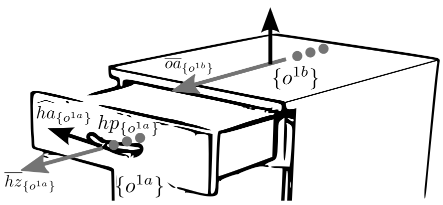
The overall application is composed of the following three states, ordered sequentially: *approach the tray* (*S.1*), *grasp the handle* (*S.2*) and *open the drawer* (*S.3*). During the overall execution, the *safety constraints* are enforced in order to limit both joint positions and joint velocities within their range. The remaining constraints are related to the specification of each state (see Table 3.5), that is:

- *Approach the tray* (S.1): the robot brings its end effector in such a position that can conveniently grasps the handle. To this end, the robotic gripper should be: *i)* oriented toward the handle, *ii)* properly rotated along its

[3]`http://stack-of-tasks.github.io/`



(a) Gripper: the line $\overline{\boldsymbol{ga}}_{\{o^2\}}$ represents the direction of approach of grasping, while versor $\widehat{\boldsymbol{gn}}_{\{o^2\}}$ must be parallel to the handle axis direction $\widehat{\boldsymbol{ha}}_{\{o^{1a}\}}$.


$\overline{oa}_{\{o^{1b}\}}$
$\{o^{1b}\}$
$\widehat{ha}_{\{o^{1a}\}}$
$hp_{\{o^{1a}\}}$
$\overline{hz}_{\{o^{1a}\}}$
$\{o^{1a}\}$

(b) Drawer: axis of the handle and direction of opening $\overline{\boldsymbol{oa}}_{\{o^{1b}\}}$ are the two main features.

Figure 3.3: Some of the geometric entities involved in the *"open a drawer"* scenario.

z-axis, and then *iii)* reducing the distance between the grasping point and the handle center to zero. Such set of constraints is shown in Table 3.5a.

- *Grasp the handle* (S.2): while the positioning constraints are held, the gripper is closing; the constraint shown in Table 3.5b is added to the task specification, while the previous are kept.
- *Open the drawer* (S.3): the handle is grasped, and the additional constraint in Table 3.5c is placed to enforce an opening motion. In the meantime, the handle must kept grasped, thus the previous constraints are not removed. However, the weighting policy of some of those can switch to *auxiliary*, since the physical coupling "gripper-handle" bind the motion anyway.

For all the above-mentioned constraints, the behaviour employed is of type *"positioning"*. Furthermore, alternative task specifications can adopt behaviours of type *"physical interaction"*; it is the case for the *"open the drawer"* specification, which can be grounded by imposing a force along the opening direction of the drawer ($\overline{\boldsymbol{hz}}_{\{o^{1a}\}}$).

The geometric expressions used to specify the monitors can differs from the ones used for the constraint specification. In details: *i)* the success of the *"approach the tray"* task can be determined by monitoring a *point-point distance* relationship between the origins of frames $\{o^{1a}\}$ and $\{o^2\}$; *ii)* the success of the *"grasp the handle"* can be visually confirmed, or achieved by a direct/indirect force measurement in the grasp space. Whereas the previous alternatives are not possible (*e.g.*, due to robot capabilities missing), it is still possible to rely on the position information (*i.e.*, current distance between the fingertips); *iii)* the

reference frame	description
$\{o^{1a}\}$	handle center, z-axis along the handle
$\{o^{1b}\}$	the chest of drawers (fixed in the world)
$\{o^{2}\}$	Tool Center Point (TCP) of the gripper
$\{o^{3}\}$	left tip of the gripper
$\{o^{4}\}$	right tip of the gripper

(a) Reference frames attached to physical objects in the scene (see Figure 3.3)

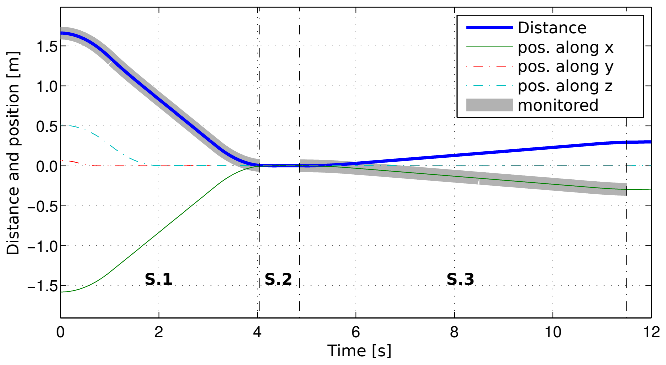
geometric primitive		type	reference frame
name	symbol		
`handle_position`	$\boldsymbol{hp}_{\{o^{1a}\}}$	*point*	$\{o^{1a}\}$
`handle_axis_direction`	$\overline{\boldsymbol{oa}}_{\{o^{1b}\}}$	*line*	$\{o^{1b}\}$, aligned with $z-$axis
`handle_axis_direction`	$\widehat{\boldsymbol{ha}}_{\{o^{1a}\}}$	*versor*	$\{o^{1a}\}$, aligned with $z-$axis
`handle_axis_x`	$\overline{\boldsymbol{hx}}_{\{o^{1a}\}}$	*line*	$\{o^{1a}\}$, aligned with $x-$axis
`handle_axis_y`	$\overline{\boldsymbol{hy}}_{\{o^{1a}\}}$	*line*	$\{o^{1a}\}$, aligned with $y-$axis
`handle_axis_z`	$\overline{\boldsymbol{hz}}_{\{o^{1a}\}}$	*line*	$\{o^{1a}\}$, aligned with $z-$axis
`grasp_position`	$\boldsymbol{gp}_{\{o^{2}\}}$	*point*	$\{o^{2}\}$
`grasp_normal_direction`	$\widehat{\boldsymbol{gn}}_{\{o^{2}\}}$	*versor*	$\{o^{2}\}$, aligned with $x-$axis
`grasping_axis`	$\overline{\boldsymbol{ga}}_{\{o^{2}\}}$	*line*	$\{o^{2}\}$, aligned with $z-$axis
`left_tip_position`	$\boldsymbol{lp}_{\{o^{3}\}}$	*point*	$\{o^{3}\}$
`right_tip_position`	$\boldsymbol{rp}_{\{o^{4}\}}$	*point*	$\{o^{4}\}$

(b) Geometric primitive definitions (see Figure 3.3).

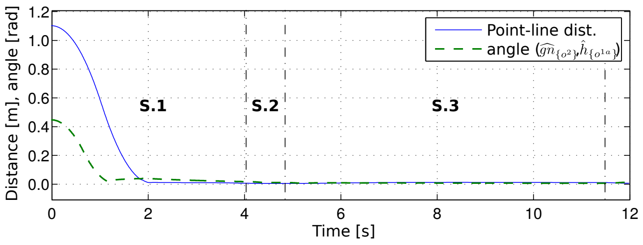
Table 3.4: Reference frames and definitions of the geometric primitives depicted in Figure 3.3 (Table 4.2a and Table 4.2b, respectively).

success of the *"open the drawer"* can be acknowledged by monitoring when the distance between the gripper position and the initial drawer position is above a given threshold. In this context, no special care is taken to specify a *failure* of a task execution; a generic timeout over the time elapsed in one state is employed. Finally, Figure 3.4 shows the monitored evolution of the expressions involved in the task specification.

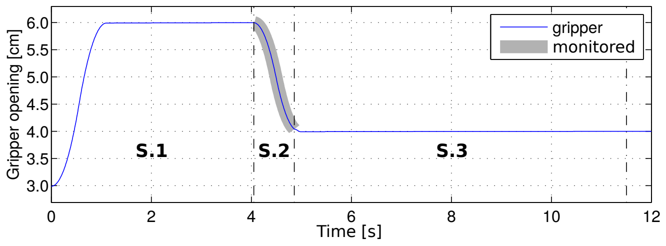
As a side note, the constraints shown in Table 3.5 are *equality* constraints. This is an ideal choice for an illustrative example, but it is often too simplistic in a real application. In fact, *i)* an equality constraint is always subject to numerical tolerances, and *ii)* a task specification based exclusively on equality constraints can easily cause an over-constrained situation or infeasibility of the solution space, while the task should be achievable. The latter is discussed in Chapter 4.



(a) x, y, z positions of gripper w.r.t. the handle (initial, fixed) frame, and total distance between frame origins.



(b) Point-line distance between the gripper grasping direction and the handle frame origin, and misalignment between the handle axis and normal grasping directions.



(c) Distance between gripper fingers.

Figure 3.4: Values of constrained and monitored expressions during simulation. Vertical lines show the transitions between states (labelled from **S.1** to **S.3**, each one corresponding to a set of *primary constraints*). In the first state, the gripper is opened (3.4c), aligned (3.4b), and brought to the handle (3.4a). A state transitions is triggered when total distance (blue flat line in 3.4a) decreases under a given threshold. During **S.2** the gripper is closed (3.4c), and, lastly, the gripper is commanded to move back to $x = -0.3\,\text{m}$ (thin solid line in 3.4a). The monitored expressions in each task are highlighted with a fat gray underling line. Furthermore, Figure 3.4c denotes a trapezoidal velocity profile that drives in feedforward the target value.

primitive I	primitive II	relation	target value
$\overline{\boldsymbol{ga}}_{\{o^2\}}$	$\boldsymbol{hp}_{\{o^{1a}\}}$	*line-point distance*	$0\,[m]$
$\widehat{\boldsymbol{gn}}_{\{o^2\}}$	$\widehat{\boldsymbol{ha}}_{\{o^{1a}\}}$	*angle btw versors*	$0\,[rad]$
$\boldsymbol{gp}_{\{o^2\}}$	$\overline{\boldsymbol{hx}}_{\{o^{1a}\}}$	*point-line projection*	$0\,[m]$
$\boldsymbol{gp}_{\{o^2\}}$	$\overline{\boldsymbol{hy}}_{\{o^{1a}\}}$	*point-line projection*	$0\,[m]$
$\boldsymbol{gp}_{\{o^2\}}$	$\overline{\boldsymbol{hz}}_{\{o^{1a}\}}$	*point-line projection*	$0\,[m]$

(a) Approach the tray *(S.1)*

primitive I	primitive II	relation	target value
$\boldsymbol{lp}_{\{o^3\}}$	$\boldsymbol{rp}_{\{o^3\}}$	*point-point distance*	$0.09\,[m]$

(b) Grasp the handle constraints, *(S.2)*.

primitive I	primitive II	relation	target value
$\boldsymbol{gp}_{\{o^2\}}$	$\overline{\boldsymbol{oa}}_{\{o^{1b}\}}$	*point-line distance*	$0.3\,[m]$

(c) Open the drawer, *(S.3)*.

Table 3.5: Constraints involved in the task specification of the *Open a drawer* scenario, divided for each discrete state.

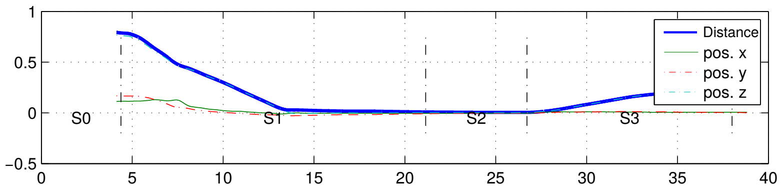
Experimental Validation

The *"open a drawer"* scenario is validated on a Personal Robot 2 (PR2) robotic platform. The overall application slightly differs from the previous example, due to the variability of the initial conditions of the experiment. For instance, a state (S.0) is added such that the task specification *"open gripper"* is executed before to start the approaching phase. As discussed in Section 2.6, the gripper must be open before it is close to the handle; since the FSM coordination model does not provide a construct for *conditional concurrency*, a conservative plan ensure the satisfaction of that logical condition.

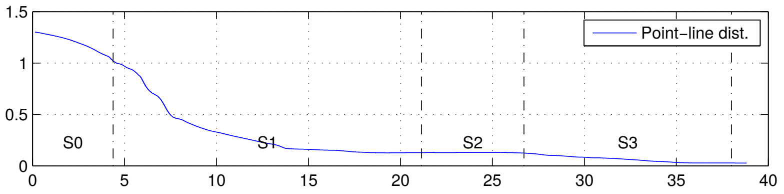
For the sake of brevity, the description of the task specifications executed in each state are not reported in this dissertation. However, the formal model, encoded as a JSON document, is available online[4]. Figure 3.6 and Figure 3.7 show a screenshot for each execution state; the integral video of the experiment can be found online[5]. The results can be appreciated in Figure 3.5: apart of the disturbances that affect the system, the outcome is comparable with the results obtained in simulation (Figure 3.4).

[4]`https://people.mech.kuleuven.be/~u0072295/jgeom_constr/examples/app/pr2-opendrawer/`

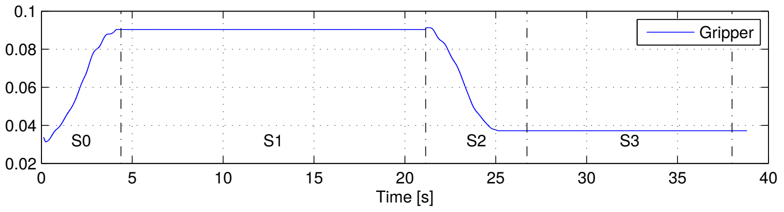
[5]`https://www.youtube.com/watch?v=4_t9dYEuswo`



(a) x, y, z positions of gripper w.r.t. the handle (initial, fixed) frame, and total distance between frame origins.

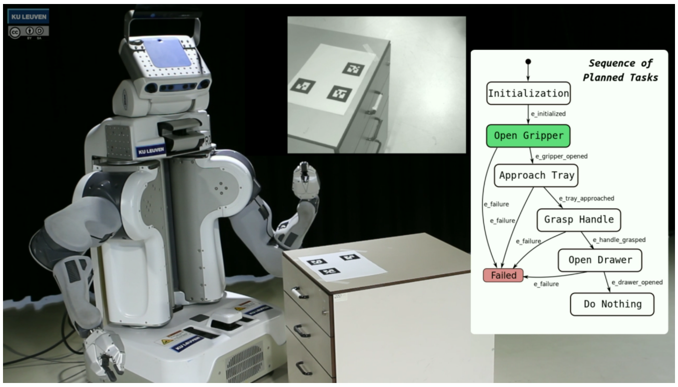


(b) Point-line distance between the gripper grasping direction and the handle frame origin, and misalignment between the handle axis and normal grasping directions.

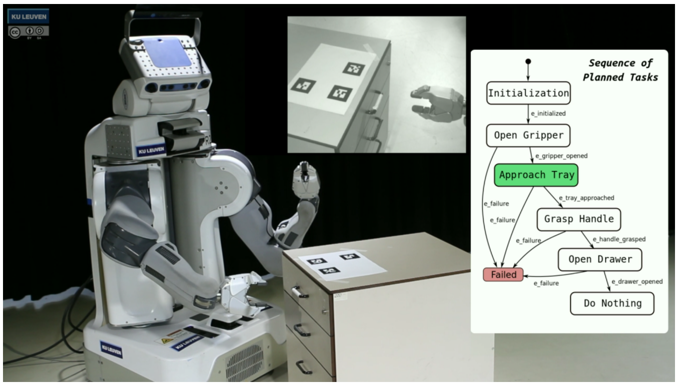


(c) Distance between gripper fingers.

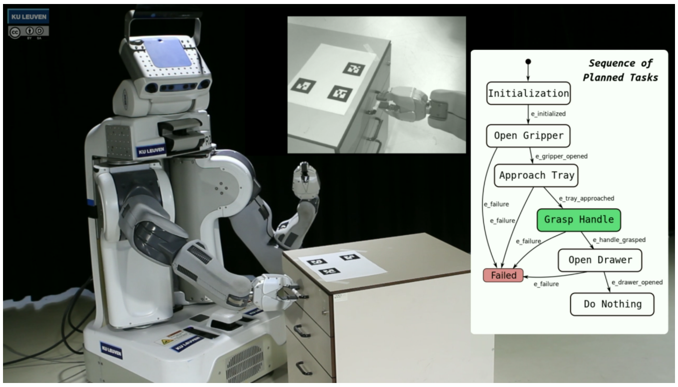
Figure 3.5: Values of constrained and monitored expressions during the experimental validation. The obtained results are comparable with the simulation of Figure 3.4. Furthermore, this experiments includes two additional states.



(a) Open Gripper (S.0).



(b) Approach Tray (S.1).



(c) Grasp Handle (S.2).

Figure 3.6: Frames of the *open a drawer scenario* sequence.

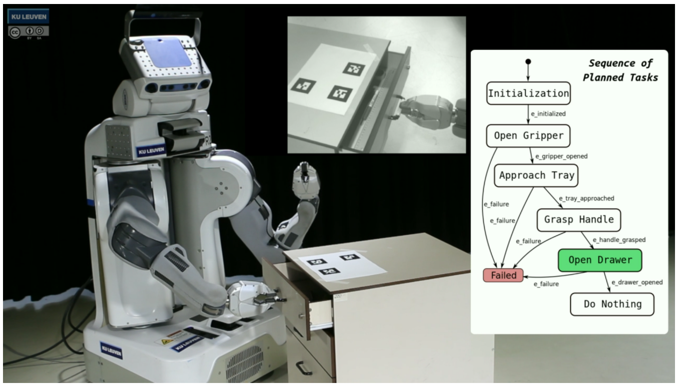


Figure 3.7: Frame from the *open a drawer scenario*, open drawer sequence (S.3).

3.3.3 Spanning a Planar Surface

Originally developed for *"spreading tomato sauce on a pizza dough"* (within the context of the *RoboHow.cog* project [128]), this example suits to other contexts that require to slide an object on a planar surface. Firstly, the robot must ensure the contact between the object and the surface; secondly, a planar motion is performed, maintaining the contact previously achieved. Thus, such tasks perform a physical interaction, which is a behaviour provided by the proposed DSL. As an assumption, the robot grasps the object, and in this case, the object is a *tool*, since it provides a particular features; examples are a spoon, a spatula and a sponge.

A validation experiment proposes a sequence of three states, each representing the execution of a different task: *i) "moving spatula above the center"* as an initial task to bring the tool in a preferred spot on the top of the target plane (*e.g.*, where the tomato sauce is piled up); *ii) "moving spatula to the center"*, task performed by combining *positioning* constraints, as well as a *"physical interaction"* constraint, which also serves as monitoring for the next action; *iii) "spread with spatula"* by means of a linear motion expressed as a positioning constraint, while the *"physical interaction"* constraint is maintained. In addition, an extra *auxiliary* constraint is imposed to maintain the arm elbow above the plane, preference that allows to impose a better physical contact. Iterating over the same set of tasks but with different initial and starting pose allows to cover different areas. Implementation-wise, the *"physical interaction"* constraint is still a scalar, since a force direction is indicated in a scalar task space, but other options are feasible. The interaction is then implemented as an admittance control scheme.

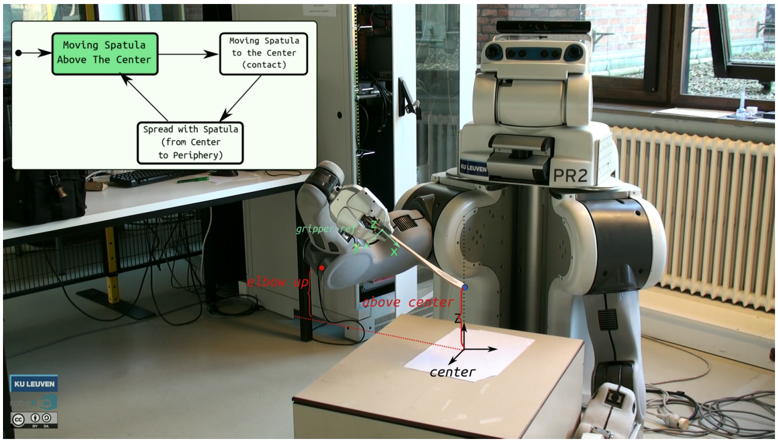

Moving Spatula Above The Center
Moving Spatula to the Center (contact)
Spread with Spatula (from Center to Periphery)
gripper_ref
elbow up
above center
center
PR2
KU LEUVEN

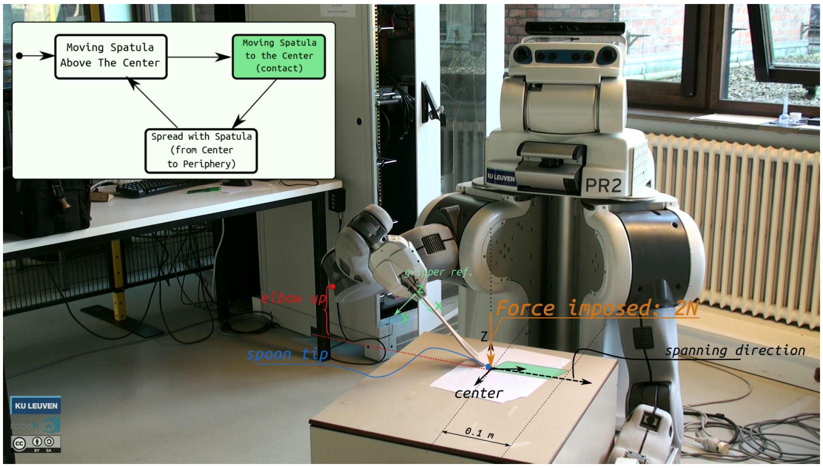
Figure 3.8: A frame from a *"spanning a planar surface with a tool"* experiment. The robot moves the spatula above the center, initial pose of the spanning task.

Screenshots of the experiment are illustrated in Figure 3.8 and Figure 3.9, while the full demo is available online[6]. For the sake of brevity, no further details are provided in this section. However, an excerpt of the task specification *"spread with spatula"* is reported in appendix A.3; the full specification is publicly available[7].

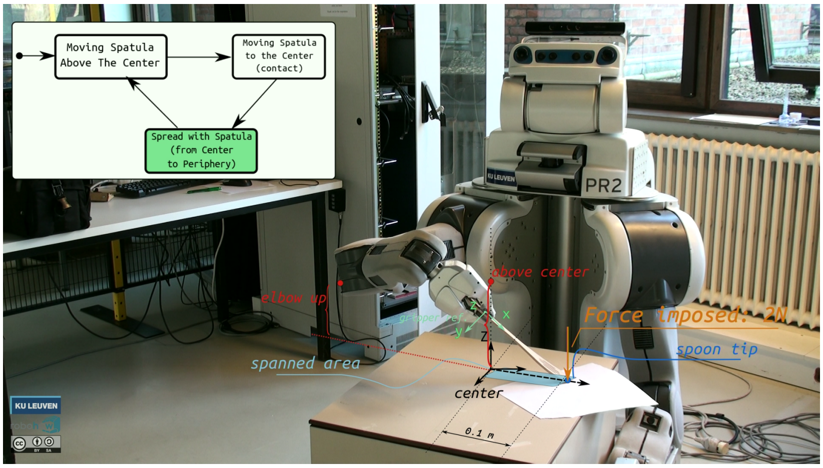
As an ending note, the above-described task specifications are executable under the assumption that the robot provides the capability of measuring the contact force on the tool tip. In the experiment, the robot relies on the feedback provided by the joint torque; neither an external force sensor or a tool model is used to estimate the force on the tip. Therefore, *the robot capability are sufficient with respect to the quality of the execution expected by the task*. The tradeoff provided by the robot capability and the concrete task is a fundamental aspect often neglected in the task specification. The proposed DSL provides a step forward in this direction: the behaviour type expresses a capability requirement, while a monitored expression within a range determines the order of magnitude of the quality expected. In this specific case of study, the capabilities of the robot have been ensured by prior work [27, 166]. For high precision tasks, the current assumption may not suffice, and better capabilities (*e.g.*, external force sensor) or extra *knowledge* (*e.g.*, a model of the grasped tool [152]) are required.

[6]`https://www.youtube.com/watch?v=Q6I4XMfl3s4`

[7]`https://people.mech.kuleuven.be/~u0072295/jgeom_constr/examples/app/spreading_spec/`


Moving Spatula Above The Center
Moving Spatula to the Center (contact)
Spread with Spatula (from Center to Periphery)
PR2
gripper ref.
elbow up
Force imposed: 2N
spoon tip
spanning direction
center
0.1 m
KU LEUVEN

(a) Moving spatula to the center (an initial pose on the plane).


Moving Spatula Above The Center
Moving Spatula to the Center (contact)
Spread with Spatula (from Center to Periphery)
PR2
above center
elbow up
gripper ref.
Force imposed: 2N
spoon tip
spanned area
center
0.1 m
KU LEUVEN

(b) Spread with spatula, linear spanning action.

Figure 3.9: Frames from a *"spanning a planar surface with a tool"* experiment. To ensure the contact between the spatula tip and the surface, a force of $2N$ is imposed orthogonally to the plane.

3.4 Conclusions

This chapter formalises a constraint-based task specification over geometric expressions. The primary aim of this work was to *unify* the various state-of-the-art approaches that deal with complex tasks on advanced robotic systems. Thus, the proposed DSL is meant as an interface between the implementation of the control strategy adopted and the entity responsible for composing the task specification (*e.g.*, a developer or an automated system). In fact, the design of the language is driven by a *composition* principle through a limited number of formally described primitives. This enables to *reason* about the task in various aspects: firstly, the formal representation permits to validate the task model against structural properties (with the help of the adopted JSON-schema); secondly, the geometric expressions must be consistent with respect to the primitives involved. Lastly, additional semantic tags provide symbolic indications on the correct formulation of the COP handled by an underlying solver implementation. In particular, the behaviour classification allows to indicate the requirements for the capabilities of both solver (software algorithms) and robot capabilities (hardware equipped).

The current implementation covers most of the common use cases, however it is limited in number of the geometric primitives and behaviour supported. Nevertheless, the presented methodology permits to extend the support to other primitives, such as curves and non-planar surfaces.

As a final note, the task specification described in this chapter is *complete* and *executable*: it contains all the knowledge required for executing the task. Some elements are *numerical*, and they must be known and fulfilled prior to execution. However, some of these are not known offline, but they are determined by an online evaluation. Thus, this task specification is not enough to describe real applications outside a protected environment as a robotic laboratory. An online coordinator must provide these values, as well as to compose the different sub-tasks, according with some symbolic constraints often described in a narrative way (see Section 3.3, *e.g.*, whether a constraint must be maintained multiple tasks): the *SDG* model of Chapter 2 is a possible solution. The next chapter will bridge those elements, providing a set of mechanisms that transforms the symbolic components to an executable task specification.

Chapter 4

Applying the Skill Dependency Graph to Constraint-based Tasks

This chapter brings together the *SDG* model of Chapter 2 and the instantaneous constraint-based task specification proposed in Chapter 3. This fusion allows to bridge the discrete description of a skill with its implementation in the continuous domain. The result is the *skill prototype* (Section 4.3), a context-dependent refinement of a symbolic action, often obtained by composition of existing ones. To deal with the variability of the context (Section 4.1), the concept of *geometric item* is introduced in Section 4.2, such that the geometric constraints that define the task are linked to instances of physical objects and their online and symbolic properties.

4.1 The Variability of the Context

Chapter 3 provided some examples of constraint-based task specification using geometric expressions. Those specifications are manually composed, starting from a known description of the environment, including the object *instances* that populate the scene. However, geometric primitives can be fully defined in the *symbolic* domain, and only grounded to a particular object later in time. It follows that the same task specification can be employed in similar *contexts*, as long as the object instances involved in the task belong to the same class.

For example, the task specification for opening a drawer is not so different from opening a door: both require a skill that allows to grasp a handle, with the aim to further interact with it. While the interaction may differ, the grasp gesture follows the same discrete pattern, whether it is grounded as behavioural-FSM or with a *SDG* model (Chapter 2); further details will be given in Section 4.3.1.

The real variability of the specification comes from the **context** in which the gesture is applied, and that depends on:

- **object instance:** the object to manipulate (*e.g.*, the handle) has some properties that constrain a possible interaction; examples are: the geometric dimensions of the object; the geometric primitives attached to the object; semantic information on the role of the object;
- **robot capabilities:** not only the robot should provide the capabilities required to manipulate the object (*e.g.*, grasping), but also the specific tool type (and relative properties) constrain the interaction in different ways; for instance, end effectors such as parallel jaws grippers, suction grippers, and robotic hands, provide different capabilities on grasping and manipulating a certain object;
- **purpose of the action:** the grounding of an action to an executable task does not only depend on the current purpose, but also on the follow-up actions; for example, the way to grasp a handle can differ if the next action is to pull, push or turn the handle;
- **environment:** the ideal action realisation may change due to environmental constraints, *e.g.*, the presence of other objects not involved in the action.

Only by combining all the above-mentioned knowledge, and solving all related constraints, it is possible to generate an executable task specification.

An initial step in this direction is given by a proper model of an object instance, which must belong to a *geometric item* class. Therefore, tasks expressed on these *classes* are a first level of abstraction from a dependent to a more independent context specification. Where the specification is still dependent on the context, the classes of items involved define the context requirements for the specification adoption. The link between these specifications and an “abstract” *SDG* model shapes a grounded model called *skill prototype* (Section 4.3).

4.2 Geometric Items

A geometric item is a model that collects the information related to a specific object, whether it is a physical or a virtual object. In general, an object model does not contain only geometric information, but it also links to other properties, such as color, weight and possible object usage; the latter model is often called *functional object* [91]. However, this section focuses only on a minimal set of properties that can be exploited in the context of a *motion* task specification, omitting other information that strictly belongs to the symbolic domain.

4.2.1 Reference Frame

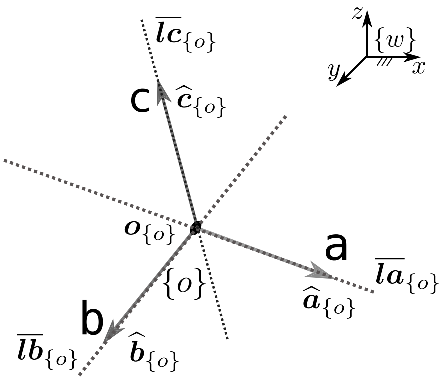
The reference frame is a *unique* attribute given to a geometric item that identifies its pose with respect to a world frame $\{w\}$, and it can be expressed in both symbolic and numerical forms. Such a property is also called *object frame* in other frameworks (*e.g.*, the iTaSC [36]) whereas the geometric item corresponds to a physical object. In this cases, the adoption of an arbitrary coordinate system **a-b-c** is motivated by the need of decoupling a generic object class from its concrete instance.

The existence of a reference frame $\{o\}$ suffices to specify the following 19 geometric primitives (see Figure 4.1a):

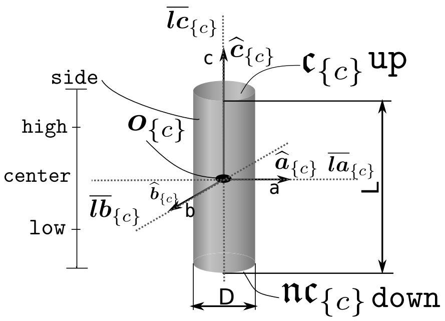
- 1 origin point, $\boldsymbol{o}_{\{o\}}$;
- 3 (+3) versors, $\widehat{\boldsymbol{a}}_{\{o\}}$, $\widehat{\boldsymbol{b}}_{\{o\}}$, $\widehat{\boldsymbol{c}}_{\{o\}}$, defined along the a,b,c-axis, respectively, and their opposite versors $\widehat{\boldsymbol{na}}_{\{o\}}$, $\widehat{\boldsymbol{nb}}_{\{o\}}$, $\widehat{\boldsymbol{nc}}_{\{o\}}$;
- 3 (+3) lines, $\overline{\boldsymbol{la}}_{\{o\}}$, $\overline{\boldsymbol{lb}}_{\{o\}}$, $\overline{\boldsymbol{lc}}_{\{o\}}$; since lines are directed, a line having opposite direction exists ($\overline{\boldsymbol{nla}}_{\{o\}}$, $\overline{\boldsymbol{nlb}}_{\{o\}}$, $\overline{\boldsymbol{nlc}}_{\{o\}}$);
- 3 (+3) planes, $\mathfrak{a}_{\{o\}}$, $\mathfrak{b}_{\{o\}}$, $\mathfrak{c}_{\{o\}}$, as well as the counter-planes $\mathfrak{na}_{\{o\}}$, $\mathfrak{nb}_{\{o\}}$, $\mathfrak{nc}_{\{o\}}$.

4.2.2 Shape Primitives

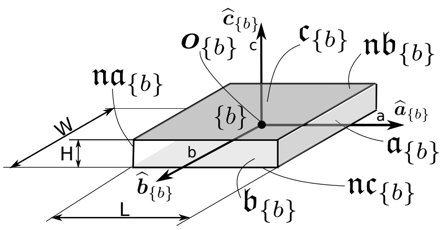
A shape primitive can be associated to a geometric item, centered with respect to the reference frame as an arbitrary convention, see Figures 4.1b–4.1d. Shape primitives are inspired by prior studies on object affordances [104, 103], and they are reduced to an extendable enumerative list, such as *box*, *cylinder*, *sphere*, *cone*



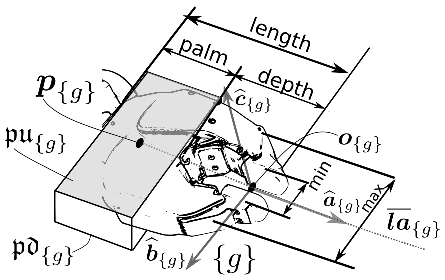
(a) Generic reference frame attached to an object ($\{o\}$) and geometric primitives generated from its origin ($o_{\{o\}}$). Planes are not illustrated.



(b) Cylindrical object $\{c\}$ and some geometric primitives generated from the model description.



(c) Box-shaped object $\{b\}$; the geometric dimensions are W,L and H.



(d) Gripper object $\{g\}$, with related geometric primitives, dimensions. The Figure is a non-exhaustive example of composition of shapes.

Figure 4.1: Geometric items.

and so on. Each shape is fully specified by a set of *geometric dimensions*, and the number of parameters depends on the shape itself. For example, a cylindrical shape (Figure 4.1b) is completely specified by the dimensions D and L, diameter and length of the cylinder, respectively; a box shape (Figure 4.1c) is determined by the dimensions W, L and H width, length and height, respectively.

A shape primitive is a convenient way to ground interesting geometric primitives for a task specification. For instance, the cylinder (Figure 4.1b) identifies two bounded planes, while the box identifies six bounded planes (Figure 4.1c). As it will be further discussed, the bounding facility provided by a shape primitive turns out to be useful to generalise a task specification. Furthermore, additional primitives can be extracted from vertices and edges of the shape.

A concrete object can be modelled as an approximation of a shape primitive, or by a *composition* of multiple shapes. The model of a parallel jaws gripper in Figure 4.1d is an example of geometric description based on a composition: the origin of the reference frame $\boldsymbol{o}_{\{g\}}$ indicates the *Tool Center Point* (TCP), while a bounding box allows to specify two palm surfaces $\mathfrak{pu}_{\{g\}}$ and $\mathfrak{pd}_{\{g\}}$. In addition, further information can be attached to the geometric model, such as the *minimum* and *maximum* span of the gripper tips.

4.2.3 Affordances of a Geometric Item

A geometric item often describes an approximation of a physical object having one (or more) functional properties; in literature, such an approach is known as *functional object mapping* [131, 91]. The main idea consists of providing semantic information (or tags) over the role of the physical item, and link such a role to a specific context. For instance, a generic box can contain some objects, but a context-dependent box can bound the contained objects to a limited subset: a tupperware stores food, while a workbox does not. These object properties, called *affordances*, compose a knowledge-space inquired by a symbolic planner, in order to generate a legal plan.

The goal of this section is not to provide an exhaustive model of functional items, but to discuss a minimal set of semantic information that allows to generalise the description of a task specification. To this end, a first distinction over a geometric item is its role in terms of *capability* provided with respect to the robotic system. Robot **resources** (sometimes called **tools**) are those that provide a certain capability to the system, and they are under control of the motion stack. For example, an end effector can provide grasping capability, as well as an interaction capability (*e.g.*, pushing an object) and sensing capabilities. Narrowing the description to grasping capabilities only, a certain tool can provide a finite number of grasp affordances, such as *pinch*, *cylindrical*, *circular*, *power grasp* and so on; a prehensile grips taxonomy can be found in [34, 124]. An additional *run-time* property indicates whether the tool is *allocated* or not, *e.g.*, if the end effector is grasping an object already, and it cannot perform another grasp until the current object is released.

Another set of geometric items are those objects that are not part of the robot, but they can be the target of an interaction with the robot. In this case, additional information attached to the geometric primitives regards the set of interactions allowed; if an interaction is possible, the related geometric primitive offers an *interaction attachment*. The set of affordances provided by an interaction attachment can be limited in number, and varying with the context. As an example, the cylinder of Figure 4.1b can approximate a glass, which can be grasped on the interaction attachment defined by the cylinder surfaces `down` ($\mathfrak{nc}_{\{c\}}$), `up` ($\mathfrak{c}_{\{c\}}$) and `side` [146]. However, if the glass is placed on a table employing the surface `down`, all the interaction attachments provided by `down` primitive are not available. Moreover, if the action to be performed is to pour something from/to the glass, the interaction attachments provided by `up` can result unavailable or limited, encouraging a grasp from the `side` primitive. In addition, the interaction attachments are not atomic, but they can be further decomposed in other semantic properties, revealing a sort of *preference*. An example is again shown in Figure 4.1b: the interaction attachment `side` provides three preferences for the grasping, that is a `high`, `center` and `low` grasping pose.

Lastly, a geometric item does not necessarily correspond to a physical object, but also to a virtual item, often called *feature*. In this case, a virtual item can aid the grounding of a task specification in several ways, such as the definition of spatial boundaries or moving targets along a geometric path.

In a robotic application, *instances* of geometric items are stored in a *world model*, and their run-time properties are updated by perception capabilities of the autonomous system. At any time, the robot can fetch some property of an object, including extra semantic information not discussed in this chapter; examples of semantic world models are [159, 46].

4.2.4 Motion Specification as a Coupling between Geometric Items

Recalling from Chapter 2, the purpose of a symbolic action is to achieve a world-transformation. Most of the actions can be grounded by describing a relationship between two (or more) geometric items, where at least one is a *tool*, thus controllable. Therefore, the grounding of a specific task specification must match with *i)* a tool that provides a certain interaction capability and *ii)* an object that allows an interaction *iii)* that suits the intended behaviour of the action. Grasping a glass with the purpose of pouring is an example: a parallel jaws gripper allows to grasp a glass from its `side`, which is an interaction that fits to a pouring action. The focus of this chapter is on actions that involve an object manipulation. However, the same principle holds for perception-driven tasks: a sensor (*e.g.*, camera) offers a perception capability suitable to extract a certain information from a target object (*e.g.*, color of the object) or the environment.

Another (sometimes implicit) feature of a robotic system is to extend (or convert) some of its own capabilities, at the expense of others: grasping an object has a temporary effect of losing a grasp capability, but other type of interactions may be provided by the grasped object. In short, a grasping feature can be seen as an extension capability, where an object is linked to a kinematic chain, turning the object into a tool. This plays an important role in the grounding of the task specification, which is further discussed in the example of Section 4.3.4.

4.3 Skill Prototypes

The introduction of the geometric items (Section 4.2) allows to generalise the construction of a task specification (Chapter 3), and to link to the *SDG* model (Chapter 2) for online composition of the constraint-based task. The overall model is called *skill prototype*, and it describes a refinement of an action, limited to the context for which it has been designed. This section is driven by concrete examples that show the design principles of a skill prototype, which is a composition of:

- a *SDG* model that represents a set of logical constraints for the online composition;
- a set of geometric expressions between geometric primitives, belonging to the geometric items involved in the action;
- a set of policies that ground each skill status to a set of constraint-based tasks based on the above-mentioned geometric expressions.

A skill prototype is not executable *as is*, but must be grounded to the specific object instances involved in the action, thus considering their geometric properties and their logical status (*e.g.*, availability of interaction attachments). In short, a skill prototype must be turned into a *skill instance*, which is a skill that fulfills the online context of its execution.

4.3.1 *Grasp an Object* Skill Prototype

A first skill prototype is a generalisation of the *"open a drawer"* scenario extensively discussed in Chapter 2 and Chapter 3. This skill separates the initial action that grasps the handle from the subsequent action of pulling the drawer. The introduction of the geometric items allows to generalise the handle as a cylindrical (or box) shape approximation; the same holds for the specific gripper, which can be mapped to a generic parallel jaws gripper model.

```
Skill: grasp,prepare, opengripper,
    approach, closegripper
Condition: close_to_object

contains(grasp,prepare)
contains(grasp,opengripper)
contains(grasp,approach)
contains(grasp,closegripper)

is-side-effect-of(approach,
    close_to_object)

d1=latches(closegripper,eff(prepare))
d2=continuesIf(approach,eff(prepare),
    close_to_object)
d3=latches(approach,eff(prepare))
d4=toStart(opengripper,eff(prepare))
d5=continuesIf(approach,eff(opengriper)
    ,close_to_object)
d6=latches(approach,eff(opengripper))
d7=toStart(closegripper,eff(approach))
d8=latches(closegripper,eff(approach))
```

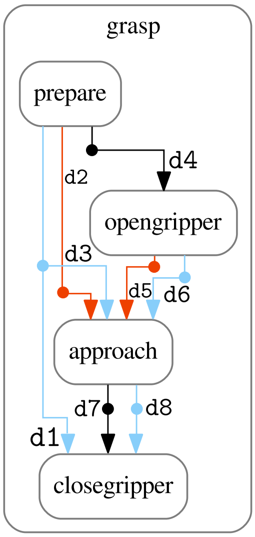


Figure 4.2: Grasp an object skill prototype; *uSDL* textual model on the left, resulting *SDG* structure on the right. Structural elements, such as ports, are omitted for the sake of brevity.

Therefore, instead of grounding the grasping action over a specific handle and gripper, a generalisation is provided by describing such an action by means of a class of geometric items: this description is suitable for grasping any object and any tool that conform to those classes.

Figure 4.2 illustrates an implemented *SDG* model composed of the following skills: *i)* `prepare` skill describes the alignment gestures for a correct pre-grasp condition, *ii)* `approach` skill implements the motion that brings the tool to the object, and finally *iii)* `opengripper` and *iv)* `closegripper` directly control the tool for the grasping. Nominally, `prepare` and `approach` skills run concurrently, as well as the skill `opengripper` once the effect of the `prepare` skill is achieved. In this way, the skill execution is optimised with respect to a conservative sequential execution.

Figure 4.3 shows the geometric expressions employed to ground the task specification, while Table 4.1 resumes the constraints that implement the skills `prepare` and `approach`. The `approach` requires an implementation of

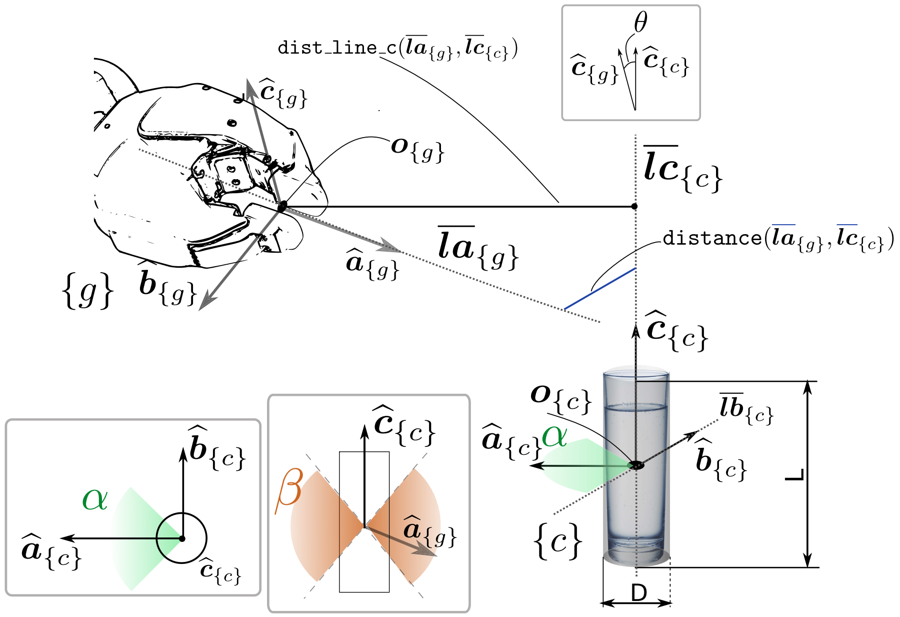


Figure 4.3: Definitions of the geometric expressions that ground the skill prototype *grasp an object*, possible refinement of the symbolic action **grasp**(*gripper*, *glass*).

the `suspending` status, here adopted as maintaining the current value of the constrained expression.

The constraints in Table 4.1a are strongly *context-dependent*, in detail:

- `align_gripper` is an equality constraint, but it can be relaxed as a pair of inequality constraints, dependently on both gripper and object models;
- `grasp_side` is expressed as a pair of inequality constraints around the length of the cylinder. However, an equality constraint can be added to express a preference over the grasping pose, previously referred as `low`, `center` and `high` interaction attachments;
- `grasp_angle_rot` and `grasp_angle_rot2` are optional constraints that depend on task, environment and object property, *e.g.*, the angle α can be limited due to the presence of an obstacle, either part of the environment or belonging to the object itself (*e.g.*, mug handle). Furthermore, the constraints can be enforced by an equality constraint instead of the adopted inequality pair, whereas necessary.

name	prim. I	prim. II	relation	target value
`dist_line_c`	$\overline{\boldsymbol{la}}_{\{g\}}$	$\overline{\boldsymbol{lc}}_{\{c\}}$	*line-line distance*	$0\,[m]$
`align_gripper`	$\widehat{\boldsymbol{c}}_{\{g\}}$	$\widehat{\boldsymbol{c}}_{\{c\}}$	*versor-versor angle*	$\theta = 0\,[rad]$
`grasp_side`	$\boldsymbol{o}_{\{g\}}$	$\overline{\boldsymbol{lc}}_{\{c\}}$	*point-line projection*	$[-L/2, L/2]$
`grasp_angle_rot`	$\widehat{\boldsymbol{a}}_{\{g\}}$	$\widehat{\boldsymbol{a}}_{\{c\}}$	*versor-versor angle*	$[-\alpha/2, \alpha/2]$
`grasp_angle_rot2`	$\widehat{\boldsymbol{a}}_{\{g\}}$	$\widehat{\boldsymbol{c}}_{\{c\}}$	*versor-versor angle*	$[-\beta/2, \beta/2]$

(a) Constraints employed in the `prepare` skill.

name	prim. I	prim. II	relation	runn. value	susp. value
`dist_line_a`	$\boldsymbol{o}_{\{g\}}$	$\overline{\boldsymbol{a}}_{\{c\}}$	*point-line projection*	$0\,[m]$	`curr`
`dist_line_b`	$\boldsymbol{o}_{\{g\}}$	$\overline{\boldsymbol{b}}_{\{c\}}$	*point-line projection*	$0\,[m]$	`curr`

(b) Constraints employed in the `approach` skill.

Table 4.1: Constraints employed for the grasp of a cylindrical geometric item on the `side` interaction attachment.

Resuming, the implementation of a skill prototype must consider the above-mentioned cases, providing an algorithm that selects the most appropriate configuration with respect to the context in which the skill is executed. To generalise the presented specification, a possible solution is to further expand each entry in Table 4.1a as a group of three constraints, one as an equality and the others as inequalities. An appropriate policy can regulate online the role of the constraints, *e.g.*, by modulation of the constraint weights based on the semantic value assigned to the tags *primary* and *auxiliary* (see Section 3.2.3).

Because of the infinite variability of a realistic application, it is not trivial to provide a skill prototype that suits to all contexts. Therefore, a skill prototype must *explicitly* define the contexts in which it can be employed, or the contexts in which cannot. The grounding reported in Table 4.1 can be adopted in case of *i)* grasping with parallel jaw grippers *ii)* of cylindrical geometric items *iii)* with interaction attachment on the `side` and *iv)* grasp of type *pinch*. Extending the previous previous cases to others is straightforward, *e.g.*, a grasp of type *envelope* can be implemented considering an internal point to the gripper than $\boldsymbol{o}_{\{g\}}$. Finally, the dimensions of the geometric items must match, *e.g.*, the diameter D of the cylindrical object must be included between minimum and maximum opening span of the gripper.

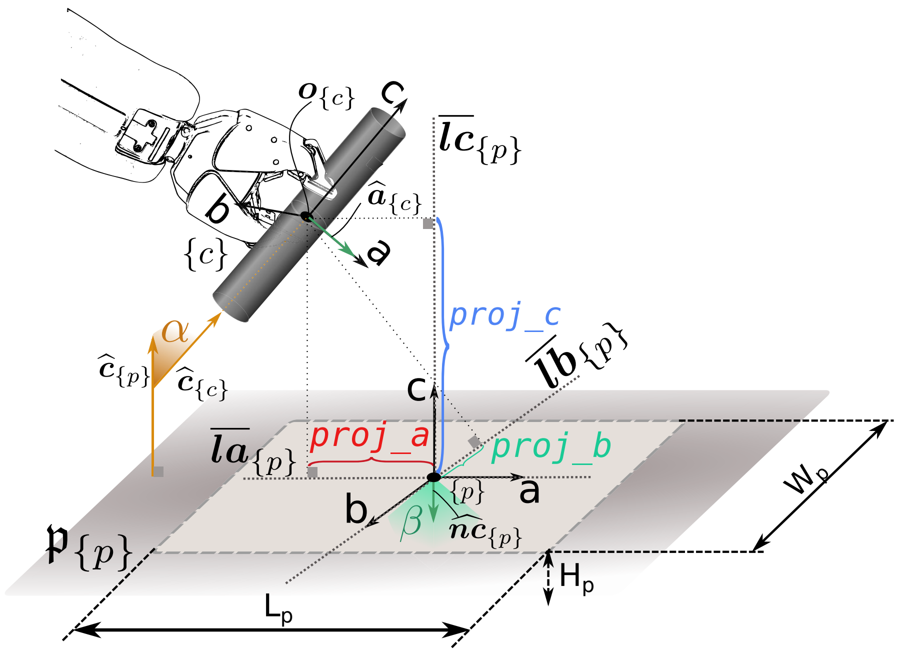


Figure 4.4: Definitions of the geometric expressions that ground the skill prototype *"place a cylindrical object on a planar surface"*, possible refinement of the symbolic action place$(glass, table)$.

4.3.2 *Place a Cylindrical Object on a Planar Surface*

Another common skill is the one that describes how to place an object on a planar surface. Figure 4.4 shows a possible set of geometric primitives (Table 4.2) and geometric expressions used as constraints (Table 4.3) to implement such a skill. In the same vein of the previous example, the imposed constraints are driven by behaviours of type *"positioning"*, which do not require any particular robot capability. However, the specification can be improved considering a *"physical interaction"* behaviour, as shown in Section 3.3.3. Furthermore, the context considered in this skill implementation regards geometric items having a cylindrical shape. In particular, Table 4.3a and Table 4.3b illustrate context-dependent solution: the former is applied in case of the intention of posing the cylinder in contact with the `down` interaction attachment; the latter in the case of contact with the `side`. The expression of the constraint `proc_c` is monitored to determine the nominal behaviour of the skill execution: if `proc_c` is satisfied, the skill is successfully executed. In addition, side-effects can be expressed upon

reference frame	description
$\{p\}$	centered on the planar surface, c-axis against gravity direction
$\{c\}$	cylinder frame

(a) Reference frames attached to the physical objects of Figure 4.4.

geometric primitive		type	reference frame
name	symbol		
`plane`	$\mathfrak{p}_{\{p\}}$	*plane*	$\{p\}$
`object_origin`	$\boldsymbol{p}_{\{c\}}$	*point*	origin of $\{c\}$
`axis_c_plane`	$\widehat{\boldsymbol{c}}_{\{c\}}$	*versor*	$\{p\}$, aligned with $c-$axis
`axis_c_obj`	$\widehat{\boldsymbol{c}}_{\{c\}}$	*versor*	$\{c\}$, aligned with $c-$axis
`axis_a_obj`	$\widehat{\boldsymbol{a}}_{\{c\}}$	*versor*	aligned with $a-$axis of $\{c\}$
`neg_axis_c_obj`	$\widehat{\boldsymbol{nc}}_{\{p\}}$	*versor*	opposite to $c-$axis of $\{p\}$
`line_c_plane`	$\overline{\boldsymbol{lc}}_{\{p\}}$	*line*	aligned with $c-$axis of $\{p\}$

(b) Geometric primitive definitions of Figure 4.4.

Table 4.2: Description of the geometric primitives adopted for the *"place a cylindrical object on a planar surface"* skill in Figure 4.4.

the same expression, such as a condition to represent whether the object is close enough to the plane. Thus, the `suspending` status of the skill can be implemented by constraining `proc_c` to maintain its current value.

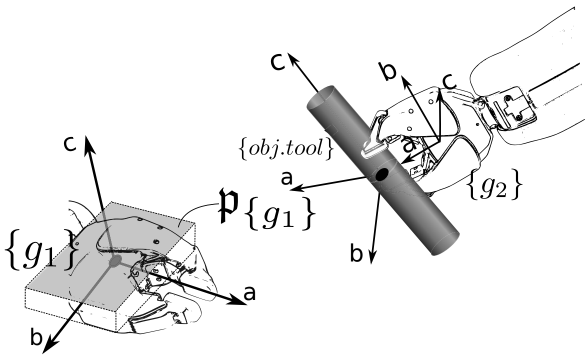


Figure 4.5: Placing a small cylindrical object to a second gripper palm.

Table 4.3c and Table 4.3d provide additional constraints imposed in case of specific requirements over both task and geometric items. In fact, this skill

name	prim. I	prim. II	relation	target value
`proj_c`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{lc}}_{\{p\}}$	*point-line projection*	$^{L}/_{2} + {}^{H_p}/_{2}$
α	$\widehat{\boldsymbol{c}}_{\{c\}}$	$\mathfrak{p}_{\{p\}}$	*versor-plane angle*	$0\,[rad]$

(a) Minimum set of constraints to place a cylindrical object on a planar surface, posed on the `down` interaction attachment, `place`$(cylinder, plane,$ `down`$)$.

name	prim. I	prim. II	relation	target value
`proj_c`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{lc}}_{\{p\}}$	*point-line projection*	$^{D}/_{2} + {}^{H_p}/_{2}$
α	$\widehat{\boldsymbol{c}}_{\{c\}}$	$\mathfrak{p}_{\{p\}}$	*versor-plane angle*	$^{\pi}/_{2}$
β	$\widehat{\boldsymbol{a}}_{\{c\}}$	$\widehat{\boldsymbol{nc}}_{\{p\}}$	*versor-versor angle*	$[-(^{\pi}/_{2} - \theta); {}^{\pi}/_{2} - \theta]$

(b) Minimum set of constraints to place a cylindrical object on a planar surface, posed on the `side` interaction attachment.

name	prim. I	prim. II	relation	target value
`proj_a`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{la}}_{\{p\}}$	*point-line projection*	$[-^{L_p}/_{2} - {}^{D}/_{2}; {}^{L_p}/_{2} - {}^{D}/_{2}]$
`proj_b`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{lb}}_{\{p\}}$	*point-line projection*	$[-^{W_p}/_{2} - {}^{D}/_{2}; {}^{W_p}/_{2} - {}^{D}/_{2}]$

(c) Optional *inequality* constraints applied if the plane primitive is bound.

name	prim. I	prim. II	relation	target value
`proj_a`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{la}}_{\{p\}}$	*point-line projection*	$L_{setpoint}$
`proj_b`	$\boldsymbol{o}_{\{c\}}$	$\overline{\boldsymbol{lb}}_{\{p\}}$	*point-line projection*	$W_{setpoint}$

(d) Optional *equality* constraints, dominating over the inequalities of Table 4.3c, applied in case of an explicit action requirements.

Table 4.3: Constraints employed to place a cylindrical object $\{c\}$ on a planar surface $\{p\}$.

refines an action of type `place`$(cylinder, plane)$, where the generic plane in not bound. In practice, a plan primitive is usually referred to a bounded surface, such as a table having known dimensions (a box-shaped geometric item), thus inequalities constraints are adopted (Table 4.3c). Furthermore, this constraints can be determined by a virtual geometric item, to shrink the area in which the object will be placed. If the action is explicit on the final pose of the object, one or both equality constraints of Table 4.3d are imposed. However, the skill prototype does not lose of generality, since it suits the non-trivial task of placing the cylinder on a gripper palm (for small enough objects), as shown in Figure 4.5.

As a last remark, the cylinder of this example must be a *tool*; it follows that

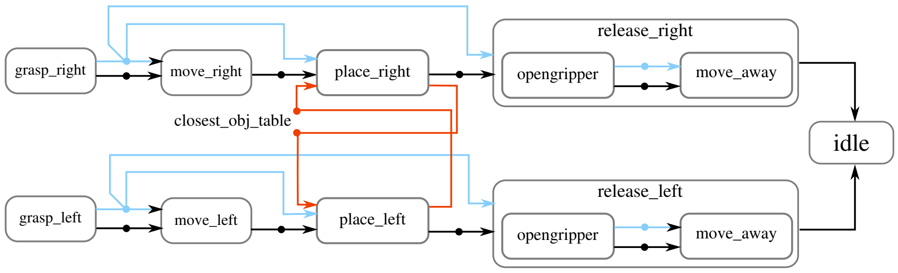


Figure 4.6: *SDG* model of a simultaneous *Pick&Place* operations, grounded as a composite skill instances of existing skill prototypes. The grasp skill expands as defined in Section 4.3.1.

```
grasp_left = {
  prototype = "grasp",
  items={ obj1="left_gripper",
        obj2="cyl1" },
  alpha = 1.57 -- [rad]
}

...

place_left = {
  prototype = "place",
  items = {
    obj1 = "cyl1",
    obj2 = "table"
  }
}
```

```
grasp_right = {
  prototype = "grasp",
  items={obj1="right_gripper",
        obj2="cyl2" },
  grasp_preference = "high",
  alpha = 1.57 --[rad]
}
...
place_right = {
  prototype = "place",
  items = { obj1="cyl2",
    obj2="table" },
  interaction = "side",
  Lsp          = 0.0, -- [m]
  Wsp          = 0.1, -- [m]
}
```

Table 4.4: Code excerpt of the additional information applied to generate a skill instance from a given skill prototype.

this skill must be preceded by another that grounds a grasping action (*e.g.*, skill prototype of Section 4.3.1).

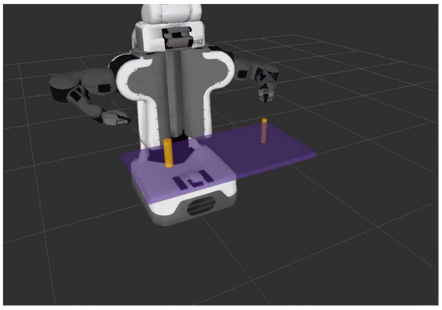
4.3.3 *Pick & Place*

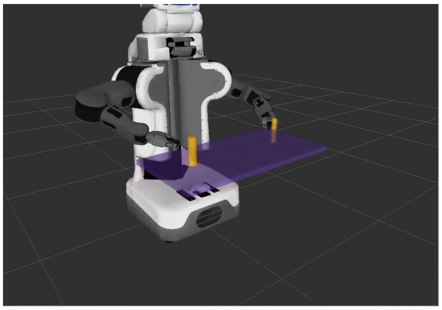
This example illustrates how the composition of few skill prototypes are enough to implement a large variability of actions. Figure 4.6 shows an instance of a *SDG* model that implements a concurrent *Pick&Place* operation executed by a dual arm manipulator.

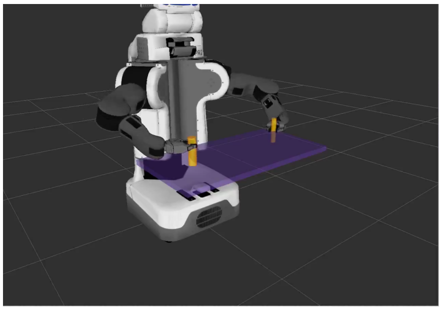
Nominally, each *Pick&Place* operation is executed independently by a single arm equipped with a parallel jaws gripper: firstly, the object is grasped, then moved to an arbitrary pose; it follows a "place on a table" skill, and finally a safe release of the object. The grasp and place skills are prototypes already discussed in the previous sections, thus only two new skill prototypes have been implemented. Furthermore, each skill instance is specialised with the additional action requirements reported in Table 4.4. In detail, the *Pick&Place* operation that involves the right arm is more specific: a grasp preference is indicated (`high`), while the `place` skill is augmented with the explicit declaration the requirements on positioning the cylinder on its `side`, and on a specific position relative to the table. Whenever no explicit information is provided, the skill instance is generated from some default values, or additional constraints are not imposed.

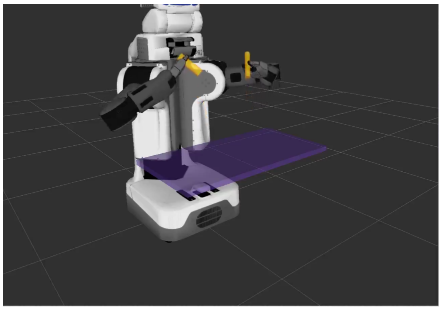
In some cases, the two independent *Pick&Place* operations may be in conflict with each other, especially if they are sharing the same workspace. A possible solution is to add an extra logical constraint of type `continuesIf` in the *SDG* model, augmented with the monitored information of which cylinder is closest to the table. In this way, a sort of priority is assigned at run-time: in case of conflicts, the execution of the skill `place` having the object farther from the table is suspended, and restored as soon as the other terminates with success. Furthermore, a skill that implements a self-collision feature is always active in this composition (not represented in the *SDG* in Figure 4.6).

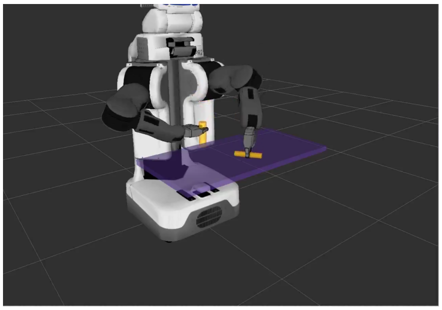
As an example, Figure 4.7 illustrates a set of screenshots of the simulated execution of the *SDG* in Figure 4.6.


(a) Initial condition


(b) Both grasp skills are `running`, (approach phase).


(c) Both approach skills are defined, and the achieved grasping follows different preferences.


(d) The cylinders are moved to an arbitrary position, simultaneously.


(e) The objects are placed considering the different preferences reported in Table 4.4.

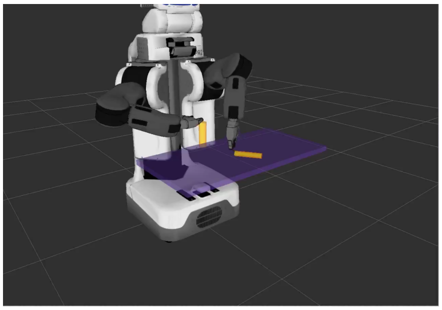

(f) The cylinders are released safely, thus the grippers are moved away from the cylinders.

Figure 4.7: Set of screenshots of the simulated execution of the *SDG* in Figure 4.4.

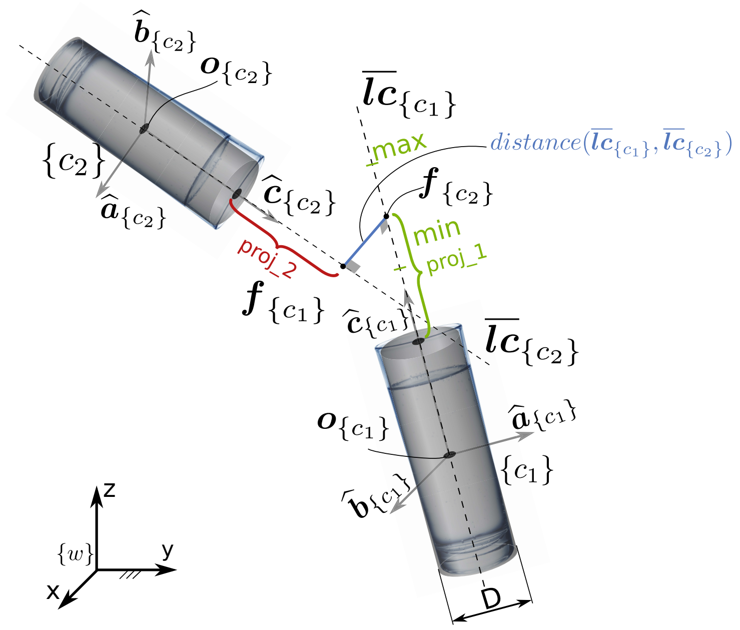


Figure 4.8: Definitions of the geometric expressions that ground a pouring skill prototype, possible refinement of the action pour($glass1, glass2$).

4.3.4 *Pouring from/to a Glass*

An interesting case of study is the grounding of an action of pouring something from a glass to another, pour($glass1, glass2$). This action is rather complex and it can be expanded in many ways, *e.g.*, depending on the robot capabilities and the overall scenario. In fact, the grounding of this action can lead to (at least) two forms: one that involves only one grasping capability (see Figure 4.9), the other that involves the coordination of two grasping resources (see Figure 4.10). The latter is more demanding, since it constrains both geometric items (the glasses) to be used as *tools* (so fully controllable), at the expenses of an additional resource (*e.g.*, a gripper), which in turn must be available. On the other hand, pouring with only one arm is less demanding from the robot capabilities perspective, but it assumes that the target glass, not controlled by the robot, is not moved or, at least, it is possible to track its motion.

The advantage of the approach proposed in this dissertation is that the execution of both strategies shown in Figure 4.9 and Figure 4.10 is obtained only with minimal changes on the *SDG* composition, while the core specification of the pouring skill is *exactly* the same. Figure 4.8 shows a minimal set of geometric expressions to implement the pouring action as a skill prototype, based on the cylindrical approximation previously discussed. The action `pour`$(glass1, glass2)$ reads as pouring to $glass1$ from $glass2$; the constraints of Table 4.5 suit to both cases in which $glass1$ is either *tool* or not, while $glass2$ must be a *tool*. The overall composition includes a skill having the effect of positioning $glass2$ on the top of $glass1$, which must be maintained during the overall pouring motion. If this constraint is not respected, the pouring motion is suspended by tilting back $glass2$, which does not necessarily constraint $\widehat{\boldsymbol{c}}_{\{c_2\}}$ being parallel to the z-axis of the world frame $\{w\}$, but enough to guarantee any further pouring.

A third, more conservative, alternative is to grasp $glass1$ first, and then to consider it not as a *tool*, but as a *regular* geometric item. In this way, $glass1$ is symbolically not constrained in the motion, and its posture is held.

reference frame	description
$\{c_1\}$	glass frame 1 (cylindrical approximation)
$\{c_2\}$	glass frame 2 (cylindrical approximation)
$\{w\}$	world frame (fixed)

(a) Reference frames in the pouring skill prototype of Figure 4.8.

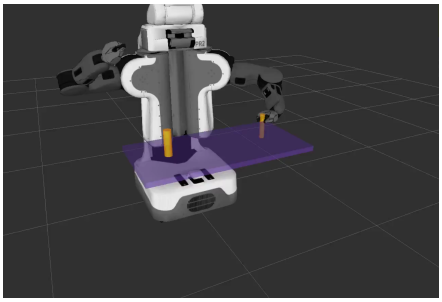
geometric primitive		type	reference frame
name	symbol		
`glass_1_origin`	$\boldsymbol{o}_{\{c_1\}}$	*point*	origin of $\{c_1\}$
`glass_2_origin`	$\boldsymbol{o}_{\{c_2\}}$	*point*	origin of $\{c_2\}$
`line_c_glass_1`	$\overline{\boldsymbol{lc}}_{\{c_1\}}$	*line*	aligned with $c-$axis of $\{c_1\}$
`line_c_glass_2`	$\overline{\boldsymbol{lc}}_{\{c_2\}}$	*line*	aligned with $c-$axis of $\{c_2\}$

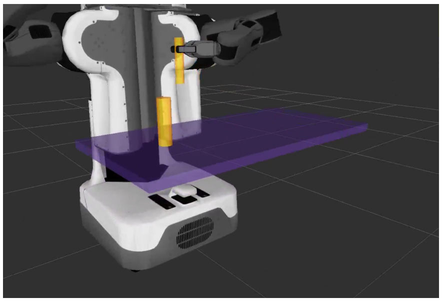
(b) Geometric primitive definitions (see Figure 4.8).

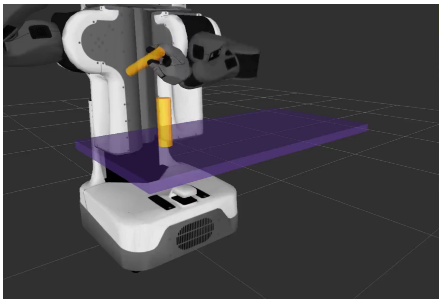
name	prim. I	prim. II	relation	target value
`distance`	$\overline{\boldsymbol{lc}}_{\{c_1\}}$	$\overline{\boldsymbol{lc}}_{\{g_2\}}$	*line-line distance*	$0\,[rad]$
`proj_1`	$\boldsymbol{o}_{\{c_1\}}$	$\overline{\boldsymbol{lc}}_{\{c_2\}}$	*point-line projection*	$[min, max]$
`proj_2`	$\boldsymbol{o}_{\{c_2\}}$	$\overline{\boldsymbol{lc}}_{\{c_1\}}$	*point-line projection*	$[0, D]$

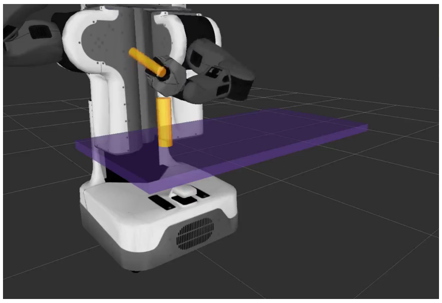
(c) Minimal set of constraints that implement the pouring skill prototype.

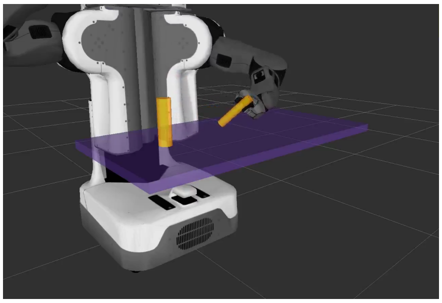
Table 4.5: Geometric expressions and constraints specification for grounding a pouring action, `pour`$(glass1, glass2)$.


(a) Grasping skill execution


(b) Moving $glass2$ on the top of $glass1$.


(c) Pouring skill in `running` status, but no content is poured yet.


(d) Pouring skill in `running` status, pouring in progress.


(e) Place skill in `running` status.

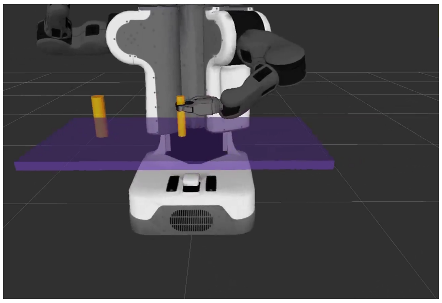
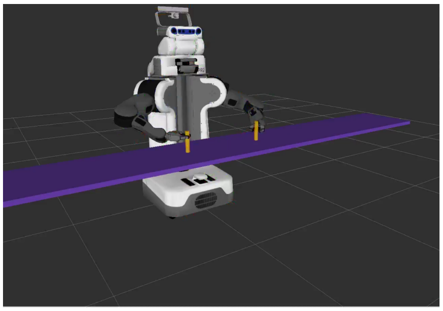
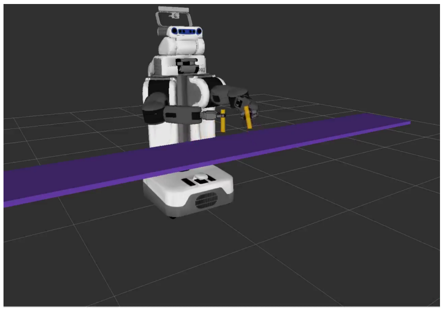
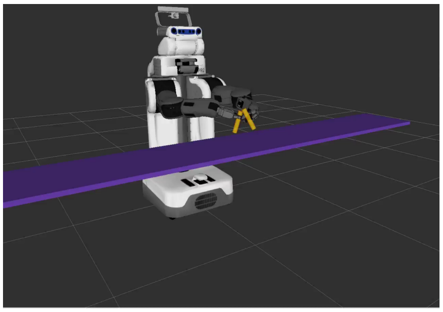

(f) $glass2$ is placed back on the table.

Figure 4.9: Execution of the composite skill that grounds the action `pour`$(glass1, glass2)$. In this case, only one grasping resource is employed, converting the geometric item $glass2$ from object to tool.


(a) Simultaneous grasp skills `running`.


(b) Preparing the pouring skill by moving *glass*2 on the top of *glass*1.


(c) Pouring skill is in `running` status, but not content is poured yet.

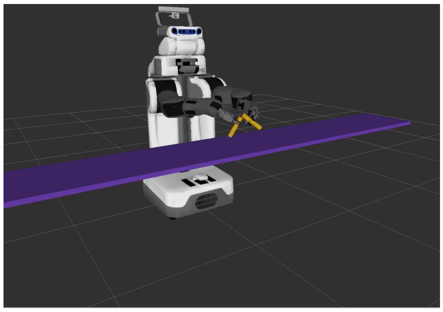

(d) Pouring skill is in `running` status, pouring in progress.

Figure 4.10: Execution of the composite skill that grounds the action `pour`$(glass1, glass2)$. In this case, both grasping resources of the dual manipulator are required such that the generated motion actively involves both glasses.

4.4 Related Work

The approach of modelling physical objects by means of their geometric features is recently emerged as a need in industrial applications. For instance, in [118] an ontology of geometric constraints is constructed from semantic annotations over existing CAD models of manufacturing objects. The latter is then exploited to define industrial tasks, see [154, 117]. However, manual annotation is not the only way to feed a model. A common solution is to extract this information from sensing data, and anchoring the recognised features for a later usage; examples are [153, 62]. The *interaction attachments* introduced in this chapter are a specific example of semantic annotations, which are directly connected to the possible usages of the object. This approach is known in planning literature as *functional object modelling* (see [131, 91, 146]). A third knowledge-based methodology is to feed the models from the web. To the best of the author's knowledge, the *KnowRob* [158, 159] is one of the most complete ontologies that cover this approach, and further improved within the context of the *RoboEarth* project [160, 164].

Anyway, the scope of this chapter was not to introduce a new ontology, nor to argue on the methods to feed the models, but to provide a minimal set of semantic information useful to generalise the description of a skill. Lastly, a preliminary work on bridging geometric features with a COP motion controller can be found in [12, 157], delivered in the context of the *RoboHow.cog* project [128].

4.5 Conclusions

This chapter proposes a methodology to merge a task specification defined in the continuous domain and a discrete *SDG* model. The result is a *skill prototype*, a refinement of a symbolic action compliant to context-dependent information. As a matter of fact, the same action can be grounded by multiple skill prototypes, which in turn can generate multiple skill instances.

Several examples are provided, showing that the composability feature of both *SDG* model and the geometric-based task specification allows to ground multiple actions with few skill prototypes, also considering and exploiting the online context of the execution.

The effort required for such a flexibility is to model and to implement the skill prototype itself, which must be developed and tested in advance by a skill expert. However, defining the requirements for which the skill prototype has (not) been designed is more relevant than delivering a prototype that aims to cover all possible cases. In fact, the latter is not feasible according to the Gödel's incompleteness theorems [127].

Chapter 5

Interleaving Planning and Execution: a Just-in-Time Approach

This chapter tackles one of the main issues that often affects many control architectures in robotics, which is the limitation of composing and configuring the overall task in a early stage of the application, or even offline. In fact, taking decisions in advance prevents optimisation and adaptation of the robotic system to context-dependent information, which is subject to changes over the time. Instead of composing a task in one shot, this chapter proposes a *"Just-in-Time"* approach applied to the *SDG* model (Section 5.2), such that decisions are made only when needed, composing the task specification at run-time, and based on the latest information available. Furthermore, some decisions that are usually part of the planning can be delegated to the execution layer; Section 5.3 shows that the *SDG* is designed to support such a feature.

5.1 Motivations

Chapter 4 introduces the concept of *skill prototype*, a description that captures and generalise recurrent patterns of grounded plans. The context in which a skill is executed provides the necessary information to generate a *skill instance* from a skill prototype. Skill instances are truly *executable* entities, since they contain both symbolic and numerical values that fulfill a task specification, which in turn configures the underlying numerical solver. However, it is not a realistic assumption to retrieve this information all at once, *e.g.*, at planning time.

Recalling the *"pouring"* skill prototype discussed in Section 4.3.4, a feasible plan for the mission *"pour off some water from a glass to another, and both glasses are placed on the table"* can be the following sequence of symbolic actions: *i)* go to the table, then *ii)* pick the glass with water, *iii)* pour the water in the second glass, and *iv)* place back the glass on the table. Already in the planning phase, some decisions can be taken on the *resources allocated* with respect to the robot capabilities. An example is the decision of executing such a plan by means of one or two robotic arms, equipped with grasping tools; this choice composes a different skill prototype that refines the symbolic action. However, other context information may not be available, such as: *i)* the gripper assigned to grasp the glass may not be compatible with the dimensions of the object (*e.g.*, because of a small opening span between the gripper tips); *ii)* the environment can cause an additional grasping constraint, *e.g.*, due to an obstacle; *iii)* if two arms are used, the decision on which arm picks which glass can be postponed; whether a criteria is established *a priori* (*e.g.*, shortest Euclidean distance between glass and gripper), its evaluation cannot be made until the pose of the glass is numerically known. Furthermore, in these cases the original symbolic plan is still valid, which is a clear hint that performing a full refinement of a symbolic action is not a responsibility of the planning. In addition, a complete refinement of the plan is not necessary to start its execution; only the first action must be fully grounded, while the others can be refined online. For instance, the `grasp` action can be refined immediately prior to its execution, or *"just-in-time"*, evaluating the context-dependent information that may cause additional constraints over the motion execution.

To resume, a single information flow from the planner to the executive does not suffice, thus multiple interactions between planning and execution are necessary. Mechanisms to interleave planning and execution layers are a research topic for any robotic architecture that aims to provide context adaptation features. The framework proposed in this dissertation is not an exception; the *SDG* model and the related plan executive implementation support a *"Just-In-Time"* (JIT) approach for such a purpose.

5.2 The Just-in-Time Approach

The term *"Just-in-Time"* is borrowed from compiler technology domain, which in turn has been inspired by the manufacturing domain. The main strategy

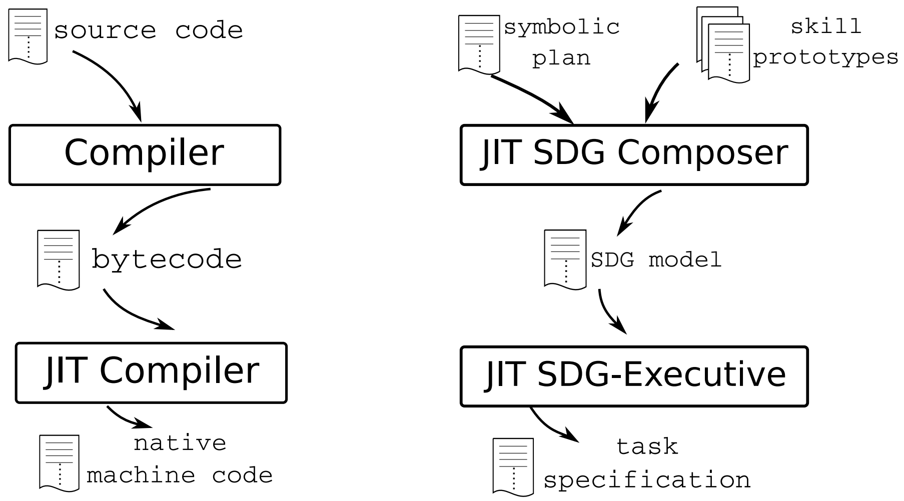


Figure 5.1: Analogy between JIT compilation (on the left) and the multistage JIT approach proposed in this section (on the right).

behind the JIT approach is simple: **making nothing, postponing decisions and evaluations until it is needed, and then producing them at the highest level of quality**. This approach goes also beyond the concepts of *lazy initialisation* and *lazy evaluation*, which are, respectively, the capability to delay the creation of an object instance and the capability to delay the evaluation of an expression. The JIT approach requires both lazy evaluation and initialisation, but it also adds the capability of translating, composing and optimizing the desired result.

In the compiler technology domain, JIT compilation translates dynamically an intermediate “temporary” representation of a program source, the so-called *bytecode*, to machine code that eventually executes the program. The bytecode is usually portable, and it decouples from the original programming language from which the bytecode has been generated. A *Virtual Machine* (VM) interprets the bytecode, optimizing whenever possible against a target CPU architecture (*i.e.*, exploiting a specific instruction set that the CPU provides, memory limitations and others). Whenever the bytecode interpretation is lazy, that is, it is not interpreted before its first invocation, the VM implements a JIT strategy.

This little detour allows to provide an analogy between JIT compilation and the JIT approach applied to a plan executive context, as shown in Figure 5.1. In detail, the *SDG*-E implementation employed in the examples of Chapter 4 conforms to a similar *meta-model*:

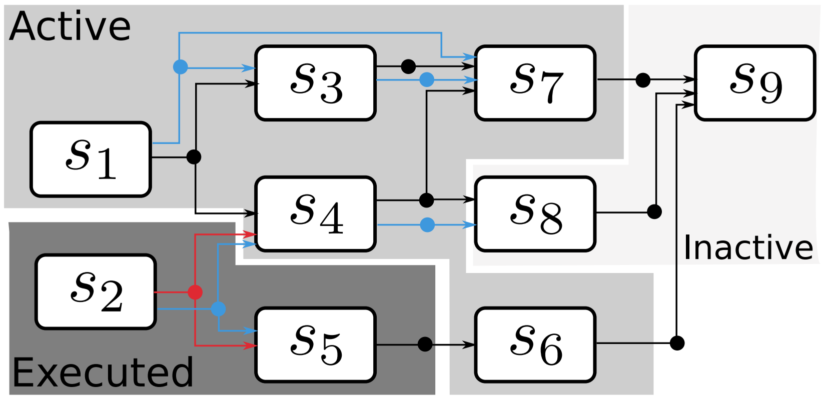


Figure 5.2: A generic *SDG* model. At run-time, it is possible to identify three sets of skills: executed, active and inactive. The *SDG* connectivity provides a metric to determine when a skill must be an *instance* and not symbolic: s_8 is next to be executed and it should be instantiated, while s_9 can be symbolic.

- *SDG* model (and the *uSDL*) is, similarly to a bytecode, an intermediate representation of a plan (the program), independently from the planner language, and it represents a first refinement necessary for its execution;
- a task specification is equivalent to the machine code of the robot, which can be optimised considering the overall context: the robot capabilities (the "resources"), the constraints given by the environment; the objects involved in the task; the action itself. A task specification is composed online by interpreting a *SDG* model, according with the (logical) constraints included in the description. Policies and capabilities that rule the COP formulation, which feeds the underlying solver, are analogous the "instruction set" (*e.g.*, constraint weight policies, priorities and so on).
- *SDG*-E is a "VM" that interprets the *SDG* model: it composes a task specification; it preserves the consistency between symbolic and numerical domain; it evaluates the symbolic variables when required for the execution.

In this analogy, the *SDG* model must be *complete* and *executable*, so it must be composed of a set of grounded *skill instances*. However, Section 5.1 explains that is not efficient to ground the whole skill in one single shot. As a matter of fact, a further JIT step can be applied when interleaving the planning and the execution of robotic tasks, involving the whole action refinement process. In detail, Figure 5.2 illustrates a generic *SDG* model and, at run-time, it is possible to identify the following sets of skills:

1. s_2 and s_5, `executed` skills;
2. s_1, s_3, s_4, s_6 and s_7 skills belong to an *active* set A, thus they are neither in `inactive` or `executed` status;
3. s_8 and s_9 are skills in `inactive` status.

The skills of the latter set are not necessarily *instances*; they can still be either a pure symbolic representation or shaped as skill prototypes not yet configured. Furthermore, the skills can be replaced, or even removed from the *SDG* composition. The constraints described in the *SDG* provide the information to decide when a skill must be grounded *in time*: at latest, this must happen when the activation constraint is satisfied. However, the refinement process can be anticipated by means of the distance between the skill and the active set A. As an example, s_8 is `inactive` but "close" to its activation (see Figure 5.2), since the activation depends only on the effect of the skill s_4, which in turn is already activated. On the other hand, the activation of s_9 is farther than s_8, due to the constraint activation dependent on the effect of s_8, which is not active yet. In short, the connectivity of the *SDG* provides a metric about the activation of the skills; Algorithm 1 reports a possible implementation (in the previous example, $\texttt{distanceActive}(s_8, A) = 1$ and $\texttt{distanceActive}(s_9, A) = 2$). This metric permits to define a criteria that triggers the initialisation of the skill, from its symbolic representation to a concrete instance, with respect to the *future horizon* of the foreseen execution.

Resuming, this second JIT step pertains to the composition of the *SDG*:

- the creation of a skill instance, starting from a skill prototype designed in advance. Such an object is statically defined, as part of "source code" required to compose an application. Since the same symbolic action can be grounded by different skill prototypes, the first step of this process is the selection of a skill prototype over the ones available, which is a context-dependent choice;
- the grounding of the skill, from the selected skill prototype to a concrete skill instance. This is a multi-stage process that involves both lazy initialisation and evaluation; more details are provided in Section 5.2.1;
- at run-time, the symbolic plan can be updated by adding, removing or finalising some actions; the implementation of this JIT composer must support this feature.

The analogy with the JIT compilation is not only conceptual: implementation-wise, the techniques adopted for the implementation are inspired by the richer set

Algorithm 1 Distance of skill s from active set in *SDG*

```
 1: function DISTANCEACTIVE($s$,$A$)
 2:     input: $s$ skill
 3:     input: $A$, set of active skills
 4:     output: $dist$, distance $s$ from active skill set
 5:     $dist \leftarrow 0$
 6:     if $s \in A$ then
 7:         return dist
 8:     end if
 9:     $C \leftarrow$ connectedTo($s$)                    ▷ set of skills connected to $s$
10:     repeat
11:         $dist \leftarrow dist + 1$
12:         $C' = \{\emptyset\}$
13:         for  $i \in C$ do
14:             if $i \notin A$ then
15:                 $s_c \leftarrow$ConnectedTo($i$)
16:                 push($C'$, $s_c$)
17:             end if
18:         end for
19:         $C \leftarrow C'$
20:     until $C = \{\emptyset\}$
21:     return $dist$
22: end function
```

of algorithms available in this domain. The major difference with JIT compilers is that motion skills require context-dependent reasoning, while compilers are typically context-free.

To the best of the author's knowledge, only a few frameworks explicitly consider formal code generation techniques to integrate and bridge different functionalities of a robotic system. An example is the *Behaviour, Interaction, Priority* (BIP) [13, 17, 1], a framework to formally describe and generate composed-based functionalities from a BIP model. However, the BIP framework focuses of model-based verification; the code generation is not treated as an opportunity to interleave planning and execution, and the approach has been demonstrated with basic motion primitives for mobile platforms only. On the other hand, the *SDG* model proposed and discussed in this dissertation aimed to concretely bridge a plan executive with a motion controller based on a COP formulation; model verification is possible, but it has not been subject of further investigations.

Similar to the JIT compilation, the proposed JIT approach is not free of limitations, which are, not surprisingly, quite similar to those of the former. The implementation of both *SDG*-E and *SDG* composer are rather complex, and it is not trivial to debug ill-formed skill instances. Pragmatically speaking, in this context bug tracking corresponds on fetching and plotting the evaluation of the expressions used to specify the constraints in the continuous domain. Such a monitoring is particularly hard, since expressions and constraints are generated online, hence it is not easy to link to them in advance. As a consequence, the workflow for implementing new skill prototypes requires an hard-testing phase to ensure the correct behaviour performance within the context in which they are applied.

5.2.1 Skill Life Cycle

This section presents the *Skill Life Cycle*, a model that represents the stages that rule the creation, execution and destruction of a skill in the JIT approach. The *Skill Life Cycle* model is not unique, and it depends on the implemented support of the underlying *SDG* composer; the adopted stages are depicted in Figure 5.3, and they are: *i) symbolic skill*, *ii) skill prototype*, *iii) skill configured*, *iv) skill instance*, *v) skill instance execution*, and *vi) skill destroyed*. Each step forward in the life of a skill instance is ruled by the set of the following operations (see enumeration in Figure 5.3):

1. **Symbolic skill creation:** the initial step to refine a symbolic action in the *SDG* model is to populate the model with an extra node that represents a skill. The node, together with an appropriate *unique* labelling that binds to the original symbolic action, suffices to constrain the skill with the others already deployed in the *SDG* model, in accordance with the given plan;

2. **Skill prototype selection:** among the skill prototypes available, one is selected to ground the action. This process involves object properties, robot capabilities and the intended effect that the skill should perform. Note that only part of the knowledge is exploited here, *e.g.*, the current pose of the geometric items is not evaluated. If the skill prototype is composite, this stage is propagated to all the children skills. Once this operation is completed, the skill previously represented in a symbolic form is linked to the chosen skill prototype;

3. **Skill configuration:** once the prototype is selected, the symbolic instances of the geometric items involved in the skill grounding are bounded as arguments to fulfill the skill description. This phase implements a *lazy initialisation* of the functionalities and resources that provides run-time

knowledge over the geometric items; examples are datastreams or services requests against perception capabilities and world model queries. Until the initialisation is not achieved with success, the skill cannot be executed; an aggressive JIT implementation can postpone the initialisation up to the satisfaction of the activation constraint;

4. **Skill initialisation:** in this stage the initialisation of the functionalities described above occurs. This process can be time-consuming, depending on the type and number of features needed, as well as the overall architecture of the system. Thus, a conservative JIT implementation must evaluate a *worst-case* scenario to avoid an excessive delay on the execution of this phase. The outcome of this operation is a *skill instance*, ready to be executed;

5. **Skill execution call:** the skill instance is called for its execution; all the expressions are well-formed and their evaluation occurs on-demand, while the behaviour is realised according with the description provided in Chapter 2;

6. **Skill destruction:** the skill is executed, resulting in either a success or a failure (`executed` or `failed` status, respectively). The skill is marked to be destroyed by a *garbage collector*, as well as the descendant skills (if any). It is garbage collector role to release all the resources not involved in the current COP definition. Optionally, a symbolic node can persist in the *SDG* model, enabling planning backtracking and historical introspection; otherwise the node is pruned away from the *SDG* model.

The operations described above are nominally executed by following the same sequence in which they have been introduced; this ordering fully describes the *Skill Life Cycle* model. However, an interaction with the planner can cause a *step back* to any previous stage in the action refinement process; examples are: an alternative choice on the skill prototype, a different geometric item involved task (*e.g.*, another object to be grasped) and so on. Furthermore, the action can be aborted, causing an immediate jump to the destruction stage. The latter is the only possible operation if the skill instance is already under execution: any step back is not possible, but only to abort the action by destroying the skill. An example of the resulting refinement process is illustrated in Figure 5.3. The *knowledge* about the current context is involved in any phase, but with a different *level of abstraction*. Thus, the design decision over the *Skill Life Cycle* influences the order of the information required, and alternatives on the proposed model are possible.

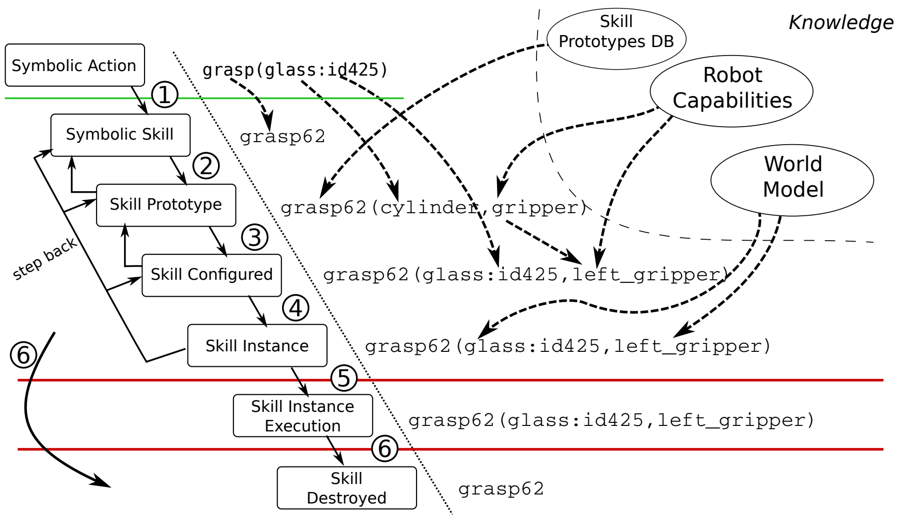


Figure 5.3: *Skill Life Cycle* stages (on the left) and a concrete example (on the right). The skill creation is described in a top-down order; full-line arrows highlight the possible stages transitions (the enumeration matches with the explanation in this section). Dashed arrows show the knowledge required to step from one stage to the next. The input is a symbolic action provided by the planner, which causes the deployment of a symbolic node in the *SDG* (`grasp62`). A skill prototype is selected from a database of skills available; such a skill must conform to the requirements imposed by the symbolic action (*e.g.*, shape of the glass, grasping tools on the robot). During configuration, the symbolic arguments (e.g., `glass:id425`, `left_gripper`) that represent geometric item instances are bound to the skill prototype; initialisation of the functionalities which retrieve online information (*e.g.*, frame pose) on the geometric item instances is postponed according with the JIT criteria over the activation distance. Once the initialisation is achieved, the skill is an instance and it can be executed by the *SDG*-E. Finally, the skill is destroyed, all required resources are released and (optionally) only the node (`grasp62`) remains for backtracking.

5.3 *SDG* Extensions: Conditional *SDG*

A complete *SDG* model represents a grounded plan composed of skill instances, and its execution ends successfully when each skill is in the `executed` status. Such a model is **deterministic**: it does not provide any support for branch predication, such as conditional constructs of type *if-then-else*, as well as control flow loops as *while-do*. Therefore, these constructs must be solved prior to execution, at composition time, since branch predications are planning primitives. This choice is not a limitation, but a design decision: the *SDG* model is a scheduling structure composed of declarative rules (the *uSDL*), which constrain *when* a skill is executed. Any decision regarding *what* to execute is a responsibility of the planner, and for this reason an online interleaving between the planning and the execution is fundamental.

The separation between *what* to execute and *when* to execute is an important research hypothesis inspired by other task-graph-based algorithms (*e.g.*, scheduling) from other engineering domains, such as compiling technologies, embedded design, multi-processor systems-on-chips and so on. In these contexts, JIT approaches are much more embraced than the Robotics domain. The current trend in the Robotics Community is to provide models and solutions that attempt to solve multiple problem in-once, simplifying the tackled issue with hidden assumptions. The latter holds for most of the state-of-the-art already discussed in Section 2.7.

However, techniques to interleave planning and execution are not trivial to implement. A solution is to delegate local decisions to the executive by means of additional information that augments the *SDG* model. This section discusses an extension to support branch predication and loop control statements, which usually reside in the planning domain. The result is a **conditional** *SDG*, also useful to design self-contained applications where the plan is provided offline by a human developer.

5.3.1 Loop Statements and Loop Unrolling

Control flow loop statements are those that indicate the need of repeat the execution of a certain action; examples are *for-do*, *while-do* and *repeat-until* constructs. In compiler technology literature, the operation that generates a finite sequence of commands from a loop statement is called *loop unrolling*; an analogous operation is adopted for composing a *SDG* model. Unrolling an action in the *SDG* consists of populating the *SDG* with as many skills instances are required for the loop termination. All the instances are of the same prototype, which grounds the action that must be repeated. Each instance differs from the others, *i.e.*, i-th skill instance operates under different world-status conditions of its previous skill instance $(i-1)$-th. Obviously, a loop unrolling is possible only if the termination condition is known and evaluable, such that the number of instances is finite and well-defined. If the latter does not hold, *e.g.*, a termination condition that depends on the outcome of the executed instance, the loop unrolling is dynamic. In general, there are two classes of loops:

- **iteration-based**: those loops that increases (or decreases) a counter for

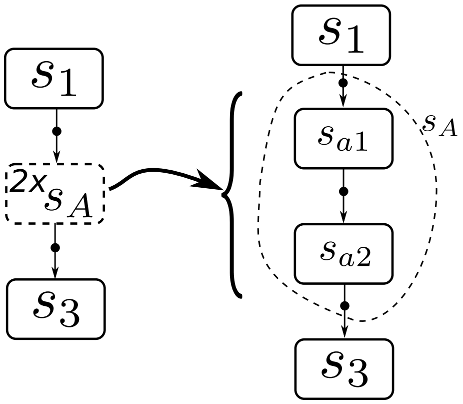


Figure 5.4: Example of a static loop unrolling. A not yet executable skill (on the left) is expanded as a repetition of the skill prototype S_A (on the right). The resulting *SDG* is composed of two instances s_{a1} and s_{a2} having a strict sequence relationship among them. An extra semantic tag (s_A, dashed line, which is a *container*, see Chapter 6) provides additional information on the origin of the generated instances; such information is needed to clean-up the *SDG* in case the planner aborts the original skill execution.

each skill executed with success. The loop is terminated if a counter threshold is met. Whereas the threshold value is statically defined, the loop is immediately unrolled as shown in Figure 5.4; otherwise the unroll process must be online, falling in the case of *external-defined condition* (see below);

- **condition-based**: those loops having an explicit termination condition evaluable online, which can be:
  - *externally defined*, *e.g.*, if its evaluation is invariant with respect to the loop execution; in this case, the evaluation occurs during the skill initialisation stage (see Section 5.2.1), and the number of repetition is determined;
  - *internally defined*, if its evaluation depends on the outcome behaviour of the skill in the loop; a first skill instance is executed and, if the execution succeeds (`executed` status), the termination condition is evaluated; if the condition does not hold, a new skill instance is appended to the *SDG*.

In both cases, the activation constraints on the unrolled skills is automatically generated, considering a strict order policy. The invariant constraints (`latches` and `continuesIf` in *uSDL*) bind to the latest skill instance of the loop.

5.3.2 Branch Predication

Planning under uncertainties requires to model branches in a plan. An example is a “pouring” action discussed in Section 4.3.4, which can be refined as a skill composition that considers the cases of *i)* grasping the glass with the left gripper, *ii)* grasping the glass with the right gripper, or *iii)* grasp both glasses (if both grippers are available). This context-dependent choice can be postponed through a proper interleaving between planning and execution.

As an alternative solution, the planner can preselect few possibilities, delegating the final choice to the executive layer. This is possible by populating the *SDG* model with the alternative skills, and inform the *SDG*-E to select one of these. To this end, the primitive `alternatives` extends the *uSDL* language described in Chapter 2.

`alternatives`$(s_1, s_2, \ldots, s_n, c_1, c_2, \ldots, c_n)$: the skills $s_1, \ldots, s_n$ are *alternatives*, and the conditions $c_1, \ldots, c_n$ are *selectors* of the skill candidate for the execution.

Whereas this primitive is applied, the *SDG* is not deterministic, but conditional; an illustrative example is depicted in Figure 5.5. However, at run-time, *one and only one skill is selected for the execution*, turning back to a deterministic situation.

```
Skill: sA,s1,s2,s3,sB
Condition: c1,c2,c3,
    ca

toStart(s1,eff(sA))
toStart(s2,eff(sA))
toStart(s3,eff(sA))
toStart(sB,ca)
alternatives(s1,s2,s3,
    c1,c2,c3)
```

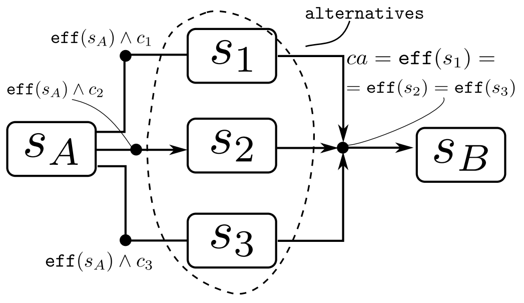

alternatives
$\texttt{eff}(s_A) \wedge c_1$
$\texttt{eff}(s_A) \wedge c_2$
$\texttt{eff}(s_A) \wedge c_3$
s_A
s_1
s_2
s_3
s_B
$ca = \texttt{eff}(s_1) =$
$= \texttt{eff}(s_2) = \texttt{eff}(s_3)$

Figure 5.5: Example of a conditional *SDG* model due to the primitive `alternatives`$(s_1, s_2, s_3, c_1, c_2, c_3)$ and its (informal) pictorial representation.

Formally, the `alternatives` primitive alterates the activation constraint of each skill involved in the relationship, that is,

$$\texttt{toStart}(s_1, c_1) \ldots \texttt{toStart}(s_n, c_n),$$

but the additional value of the `alternatives` primitive is provided by the constraints that determine the well-formedness of the declarative rule, which are:

- *selectors are mutually exclusive*, formally
$$\texttt{holdsAt}(c_1, t) \rightarrow \neg\texttt{holdsAt}(c_2, t) \wedge \cdots \wedge \neg\texttt{holdsAt}(c_n, t); \tag{5.1}$$
- *Control Flow Uniqueness:* the tail of each branch must produce the same effect, that is
$$\texttt{eff}(s_1) = \texttt{eff}(s_2) = \cdots = \texttt{eff}(s_n). \tag{5.2}$$

Resuming, the decision making delegated to the *SDG*-E is implemented as a regular evaluation of extra conditions (the selectors), and their design must guarantee the above-mentioned constraints. The selector concept permits to further extend the *SDG* from conditional to *probabilistic* graph, in those cases where the evaluation of the condition is associated to probabilistic values, *e.g.*, perception information with a certain degree of uncertainty. However, no further investigations are made in this direction, highlighting a possible future work in this direction.

Lastly, the `alternatives` primitive as defined above involves only the activation dependency constraint. Similar derivations are possible considering invariant-based constraints, for instance, by grouping alternative skills with mutually exclusive guard conditions. This suggestion allows to introduce other features, such as a continuous switching between alternative skills.

5.4 Conclusions

The contributions of this chapter are manifold: firstly, a JIT methodology has been introduced and applied to a robotic context which is, to the best of the author's knowledge, a first attempt of this kind; secondly, a concrete skill-based model to refine a symbolic action to its implementation has been proposed. Together, these efforts permit to postpone context-dependent evaluations that influence the definition of the task specification. In this way, the task specification is optimised and adapted to its execution context. Lastly, it has been illustrated how to delegate symbolic decisions, usually made in the planning phase, to a local execution layer based on the *SDG*.

This chapter did not focus on the technicalities of the sofware implementation, but on the formalisation of the concepts that, hopefully, will contribute to shape the future of robotic programming.

As a last remark, the software that executes the examples of Chapter 4 has been implemented with the technical notes provided in this chapter.

Chapter 6

Hierarchical Hypergraphs and the NPC4 Domain Specific Language[1]

6.1 Introduction

Many robotics applications rely on *graph models* in one form or another: perception via probabilistic graphical models such as Bayesian Networks; control diagrams and other computational “function block” models; software component architectures [31, 167]; Finite State Machines [81] (*e.g.*, Figure 6.1); kinematics and dynamics of actuated mechanical structures [142]; world models and maps [22, 21]; knowledge relationships as the so-called *RDF triples* (Resource Description Framework), and so on. In traditional graphs, each edge connects just two nodes, and graphs are *flat*, that is, a node does not contain other nodes.

This chapter is a shorter version of the work submitted in [138], which introduces *hierarchical hypergraphs* as an alternative to traditional graph models: *i)* an edge can connect more than two nodes, *ii)* the attachment between nodes and edges is made explicit in the form of “ports” to provide a *uniquely identifiable* view on a node's internal behaviour, and *iii)* every node can in itself be another hierarchical hypergraph. These properties are encoded formally in a *Domain Specific Language* (or “a meta-model of a language"”), called “*NPC4*”, built with *node*, *port*, *connector*, and *container* as primitives, and *contains* and *connects*

[1]The content of this chapter is based on the research presented in [138].

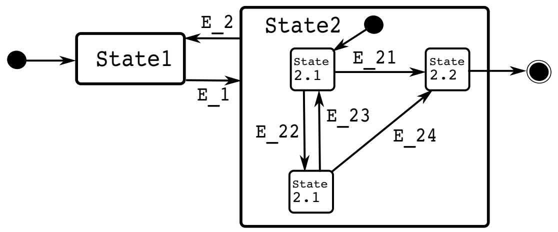


Figure 6.1: A hierarchical Finite State Machine. *Nodes* represent states, and *edges* represent state transitions.

as its *composability* as a meta modelling language, for both the *structural* and *behavioural* parts of more concrete (Domain Specific Languages) (DSL) that can be built on top of it, each in a specific domain *context*. *NPC4* introduces a particular primitive, the *container*, to support *overlapping contexts*. It suits to the following major targets in *knowledge-centric* robotics systems: (i) various *levels of abstraction* in domain models, (ii) "multiple inheritance" from (or rather "conformance to") different knowledge domains, and (iii) connecting one or more domain DSLs to the same *software infrastructure* in which they all have to be "activated".

The **central research hypothesis** of this work is that the concept of the *hierarchical hypergraph* is a good formal representation to cover *all* the compositional structures mentioned above, more particularly, via the *property* (*"has-a"*), *containment* and *connection* (*"interacts-with"*) primitives. Obviously, each application domain needs more than only a *structural* model; the approach in this work makes sure that composability is a first-class design driver.

Core idea and objectives: the aim is to provide better modelling flexibility and methodology to robot system developers, by introducing them to a hierarchical hypergraph *meta-meta-model*. The *NPC4* meta-model represents the *structural* properties of all the use cases introduced before, in a fully formal, computer-processable way, and with a clean *separation between structure and behaviour*. *Structure* models which subsystems interact with which *other* ones, and how their *internal* structure looks like; and *behaviour* models the "dynamics" of each of interconnected subsystems, and how the interconnections influence those subsystem dynamics.

The **research hypothesis** that *NPC4* provides:

- a *separation* between structure and behaviour (or functionalities);
- a methodology of making a new DSL by *only* having (i) *to specialize* the interpretation of *NPC4*'s primitives (node, port, connector, container)

to the domain, and (ii) *to add* constraints to the *contains* and *connects* relationships.

- the minimal set of language primitives and relationships that supports all DSLs in the robotic domain.

In addition to the envisaged optimal *reuse of modelling concepts*, the systematic approach is also expected to create a step change in *reuse of software*:

- *reuse of syntactical parsing code*: the structure of a DSL is visible through the language's syntax, and since *NPC4* provides a common structural basis to DSL builders, they should be able to reuse a lot of the parsing software.
- *reuse of infrastructure code*: every DSL that is being introduced in a robotics system requires more support from the system's infrastructure code than only the realisation of the modelled domain functionalities, *e.g.*, logging, messaging, debugging, tracing, and so on. *NPC4* provides all the "hooks" to connect these non-functional software requirements to.
- *reuse of "Model-to-X" transformation tooling*: models are *declarative* specifications of domain functionalities, and inevitably needs to be *transformed* into code that supports turning the declarative specifications into procedural code, and basing different DSLs onto the same *NPC4* core simplifies reuse of such model transformation tools.

Due to the heterogeneous types of applications that can benefit from the introduction of hierarchical hypergraphs, the results presented in this chapter were obtained by shared efforts of many researchers. The author of this dissertation contributed to formalise the *NPC4* model, as well as to realise the cases of study presented in [138].

For the sake of brevity, the content of this chapter is limited on the *Skill Dependency Graph* (*SDG*, Chapter 2), which is one application example; insights and examples in broader contexts are not discussed in this work, but they can be found in [138]. Section 6.2 explains the semantics of what this work understands under the term "hierarchical hypergraph". It also creates a fully formal language for hierarchical hypergraphs, the *NPC4*, in the form of a DSL. Section 6.3 presents constraints and properties over the *NPC4* language, while Section 6.4 discusses about composability features of the language. Section 6.5 illustrates that the *SDG* model conforms to the *NPC4* language: the structural primitives are enriched with domain-specific constraints regarding the well-formedness of a *SDG* model.

6.2 Hierarchical Hypergraphs

This section proposes the adoption of *hierarchical hypergraphs* in the robotics domain, instead of traditional graphs, as its main structural model. The motivation is based on the list of examples in Section 6.2.1 that illustrate various ways in which the use of traditional graphs introduces erroneous ways of representing and reasoning about complex systems. Many users of graph models are not aware of the hidden assumption of the specific domain, or cannot formulate them by lack of an appropriate and semantically well-defined language; the primitives of such a language, *NPC4*, are introduced in Section 6.2.2, and then formalised in Section 6.2.4.

6.2.1 Motivations and Bad Practices

Traditional graphs have *nodes* and *edges* as model primitives, and most practitioners feel very comfortable with using them as graphical primitives for modelling. However, traditional graphs have a rather limited expressivity with respect to modelling the *structural* properties of a system design. The paragraphs below explain commonly occurring "bad practices" in using traditional graphs.

An edge can only connect two nodes, while many structural interactions are so-called *n-ary relationships*, that is, more than two (*i.e*, "*n*") entities interact at the same time, and influence each other's behaviour. A *SDG* model exhibits such relationships as first-class citizen primitives, and domains having a constraint over the number of connections that an edge can host must be explicitly expressed.

The structural model is flat, in that all nodes and edges in the model live on the same "layer" of the model. However, *hierarchy* has, since ever, been a primary approach to deal with complexity in design problems by allowing to interconnect various *levels of abstraction* when modelling a system. For example, a *kinematic* model of a robot structure might be enough for motion planning, but the *dynamics* of its actuators might be needed to design the robot's motion controllers. Since the actuators are mechanically connected to the kinematic chain of the robot, a hierarchical structural model would apply perfectly to support the separation between the kinematic and dynamic models of the same robot. Also *knowledge relationships* are prominent examples of where the problem of flat structural models is very apparent: here, *hierarchy* is equivalent to *context*, that is, the meaning of a concept depends on the context in which it is used. Context is an indispensable structure in coping with the information in, and about, complex systems. A third prominent "bad practice" example of "flat" structural models are the popular (open source) *robotics software frameworks*, like ROS [126] or Orocos [29]: they do not support hierarchical composition of software components, the consequence being that users always see all the dozens, or even hundreds, of nodes at the same time. This makes understanding, analysis and debugging of applications difficult.

Interactions are uni-directional. Most modelling approaches use *directed* edges, that is, the graph assumes that each "partner" in an interaction can influence one or more other "partners", without ever being influenced itself by those partners in any way. Nevertheless, *bi-directional* interactions are the obvious physical reality: interactions, including man-machine interactions, exchange energy in both directions.

6.2.2 Primitives, Relationships and their Semantics

This section introduces a minimal and complete set of primitives and relationships to describe a semantically consistent structural model. The concepts of *hyperedges* and *hierarchy*, as key additions to existing graph modelling traditions, aim to prevent the implicit, domain-specific assumptions previously discussed.

The core of the language are the **structural relationships** of `has-a`, `connects` and `contains` between the **model primitives** of `Node`, `Port`, `Connector` and `Container`. The semantic role of a `Node` is to host a behaviour, while a `Connector` describes the interaction relationship between the dynamics inside *multiple* `Nodes` by "connecting" them. Formally, a `Connector` realises an *hyperedge*, since the relationship is not *unary* but *n-ary*, and is bi-directional by default (that is, unless explicitly constrained not to be so.) In traditional graph modelling, a duality property exists between `Node` and `Connector`: both can be seen as *vertex* or *hyperedge.*

However, this symmetry disappears as soon as the *containment* relationship is introduced. In fact, the *hierarchy* concept is orthogonal with respect to the *hyperedge connection* concept. *Hierarchy* is expressed by the relationship `contains` applied to the `Node` primitive: a `Node` can contain a full hierarchical hypergraph in itself. The latter is semantically justified by observing that the hosted behaviour by the `Node` can be structurally represented as composition of internal `Nodes` and the interactions between them. Note that *composition* is a primary design driver of the proposed hierarchical hypergraph approach.

To achieve full expressiveness of the structural model, the `Port` is formally introduced as the third primitive in the language. A `Port` offers a specific *view* of a `Node`, exposing a specific part of a `Node`'s internal behaviour, and creates structure in the *connects* relationships across *hierarchy* levels. As a consequence, the `connects` relationship involves directly the `Port` primitive, and not `Nodes`, as it will be illustrated in the following section.

Finally, a primitive called `Container` provides a grouping feature, allowing to add extra semantic knowledge to a selected subset of primitives; such grouping is known under various names, such as “context”, “namespace” and “scope”.

6.2.3 Design Drivers

The major design drivers to ground the *hierarchical hypergraph* concepts as a *Domain Specific Language* are *minimality*, *explicitness* and *composability*:

Minimality. The model represents only *interconnection and containment structure.* It serves as skeleton to represent the information about the structural model, but it does not make any assumption on the behaviour of such a structure.

Explicitness. Every concept, and every relationship between concepts, gets its own explicit keyword:

- `Node` for the concept of behaviour *encapsulation.*
- `Connector` for the concept of behaviour *interconnection.*
- `Port` for the concept of *access* between encapsulated behaviour and each of its interconnections.
- `Container` for the concept of *packaging* a model in an entity that can be referred to in its own right.
- `contains` for the relationship of composition into *hierarchies.*
- `connects` for the relationship of composition via *interaction.*

Composability. The DSL is intended to represent only *structure*, and is, hence, *designed* to be extended (or *composed*) with *behavioural* models: it allows to connect other models to *any* of its own language primitives and relationships, *without* having to change the definition of the language (and hence also its parsers or other supporting software and tooling).

6.2.4 Formalisation into the *NPC4* Language

The previous section provided an overview about the role and the motivations of the primitives and relations proposed in this work. This section turns this into

Primitive \ Primitive	`Node`	`Port`	`Connector`	`Container`
`Node`	contains	has-a	contains†	contains
`Port`	part-of	-	connects	is-contained*
`Connector`	is-connected (port)$^+$	connects	-	is-contained*
`Container`	contains	contains	contains	contains

Table 6.1: Overview of the primitives introduced by *NPC4* and the relative structural relationship allowed between them. The table reads has `{primitive-row}` `{relationship}` `{primitive-column}`, *e.g.* "a `Node` (can) contains a `Node`". Notes: (i) * it is not a relationship in *NPC4*, passive form; (ii) $^+$ it is not a formal relationship in *NPC4*, but informally a `Connector` is *indirectly* connected to a `Node` through a port; (iii) † as property of a *well-formed* `Connector`, see Section 6.3.1.

a concrete DSL, the NPC4 *meta-model for hierarchical hypergraph*. Considering the modelling layers suggested in [8, 113] (see Chapter 1.3.4), the *NPC4* DSL resides on M2 level, and its domain is the description of hierarchical hypergraph structures. Since this work focuses more on the ontological/theoretical model rather than a linguistic one, the M3 *meta meta* model for *NPC4* is the mathematical foundation of a *graph*. A further assumption is that multiple *meta-models* on M2 can coexist and can be composed[2] into new DSLs. In this cases, a new DSL can *conform to* one or more other meta-models on M2 level.

The *textual* formalisation of the language is discussed, while Figure 6.2 shows the corresponding *graphical* conventions adopted. Table 6.1 provides an overview on the language core, and Table 6.2 illustrates the DSL by means of the concrete example of Figure 6.3.

`Identity` is given to all primitives by simple declaration:

$$\texttt{Node}: \quad \texttt{node-B}, \texttt{node-X}, \ldots \tag{6.1}$$

$$\texttt{Port}: \quad \texttt{port-p}, \texttt{port-x}, \ldots \tag{6.2}$$

$$\texttt{Connector}: \quad \texttt{connct-i}, \texttt{connct-j}, \ldots \tag{6.3}$$

$$\texttt{Container}: \quad \texttt{cntnr-m}, \ldots \tag{6.4}$$

Furthermore, let `{node}`, `{port}`, `{connector}` and `{container}` be the sets of all the declared `Node`s, `Port`s, `Connector`s and `Container`s, respectively.

`has-a`: a relationship between a `Node` and a `Port`. A `Port` can exist on itself (*e.g.*, when it is still "floating" during the construction of a graph model in a development tool), but the graph model can only be "well-formed" (see

[2]This composition of DSLs is sometimes denoted as language *mixin*.

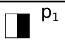
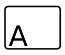
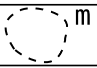
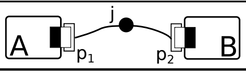
Primitive	Graphical Convention
Port	
Node	
Connector	
Container	
Connection Node-A Node-B	

Figure 6.2: Graphical conventions to represent hierarchical hypergraphs: (i) `Port` is a square composed of two rectangles which represent (with respect to the `Node` to which the `Port` is attached) the internal (black) and external (white) docks; (ii) a `Node` is a rounded box; (iii) the `Connector` is shaped as a filled circle; (iv) the `Container` is represented as a dashed outline. The bottom row shows an example of two `Nodes`, namely A and B, connected by the `Connector` j attached to the external docks of `Ports` p_1 and p_2. The "clamps" on the docks appear if the docks have been linked to a connector.

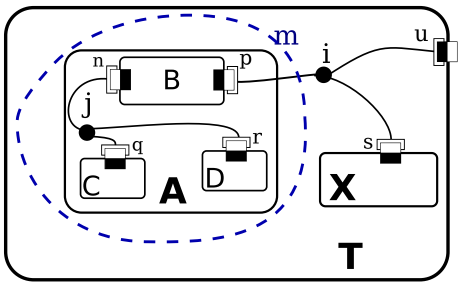


Figure 6.3: Generic example of a hierarchical hypergraph model. `Node` T is at the top of the hierarchy, and allows to refer to the whole model from within other models. `Nodes` A and X are contained by T, as is `Container` m; `Nodes` B, C and D are contained by A. `Connectors` i and j link `Ports` on `Nodes`. All `Ports` have `Connector` docks internal and external to the `Node` they belong to. `Container` m gives a context to `Node` A and its internals, but not to `Node` X or `Connector` i.

Section 6.3) if every port belongs to exactly one node. Ports are those parts of a node through which (a *selected subset* of) the latter's behaviour becomes *accessible for interaction* to other nodes. So only statements of the following type make sense:

$$\texttt{has-a}(\texttt{node-B}, \texttt{port-p}), \tag{6.5}$$

```
Node: node-A, node-B, node-C, node-D, node-X, node-T
Port: port-q, port-r, port-p, port-n, port-u, port-s
Connector: connector-j, connector-i
Container: container-m

has-a(node-T, port-u)
has-a(node-X, port-s)
has-a(node-B, port-n)
has-a(node-B, port-p)
has-a(node-C, port-q)
has-a(node-D, port-r)

contains(node-A, node-B)
contains(node-A, node-C)
contains(node-A, node-D)
contains(node-T, node-A)
contains(node-T, node-X)
contains(container-m, node-A)
contains(container-m, connector-j)

connects(connector-j, port-q.edock)
connects(connector-j, port-n.edock)
connects(connector-j, port-r.edock)
connects(connector-i, port-p.edock)
connects(connector-i, port-s.edock)
connects(connector-i, port-u.idock)
```

Table 6.2: Full *NPC4* model of the example shown in Figure 6.3.

and statements of the following type *do not*:

$$\texttt{has-a}(\texttt{connct-i}, \texttt{port-p}), \texttt{has-a}(\texttt{cntnr-m}, \texttt{port-p}).$$

The inverse relationship `part-of` could be added to the model language, as syntactic sugar:[3]

$$\begin{aligned} & \texttt{part-of}(\texttt{port-p}, \texttt{node-B}) \\ \Leftrightarrow \quad & \texttt{has-a}(\texttt{node-B}, \texttt{port-p}). \end{aligned} \tag{6.6}$$

`has-a`: a second relationship of this kind exists between a `Port` and a `dock`. The `dock` is a structural property of the `Port` that holds at most one connection

[3]Informally, in this work the following sentences are equivalent of expressing an `has-a` relationship: (i) "a port belongs to a node", (ii) "a port is attached to a node".

with a `Connector`. Each `Port` has exactly two `docks`, one *internal* and one *external* with respect to the `Node` which owns the `Port`. The `docks` are true `Port` properties by design, therefore they are not considered as a primitive of the language. To distinguish with respect to the previous `has-a` relationship, the `dock` is uniquely referred by a *dot* (`.`) notation, that is:

$$\forall \mathtt{P} \in \mathtt{\{port\}}, \exists! \mathtt{P.edock}, \exists! \mathtt{P.idock} \tag{6.7}$$

with `edock` and `idock` being a port's external and internal dock, respectively. The `dock` property will turn out to be important later on, when well-formedness of connectors will be discussed in Section 6.3.

Figure 6.2 shows the graphical convention of a `Port`, visualised as box divided in black and white rectangles; the former represents the *internal* `dock`, the latter is the *external* `dock`. The `has-a` relationship between `Node` and `Port` is visualised by placing the `Port` along the `Node` border.

`contains`: `Nodes` and `Containers` can contain other primitives, as represented by containment statements of the following type:

$$\mathtt{contains}(\mathtt{M}, \mathtt{X}), \tag{6.8}$$

with `M` and `X` being a `Node` or a `Container`. The `contains` relationship brings *hierarchy* in the relations between `Node` and `Container` primitives.

Containment is a *transitive* relationship, so other containment relationships can be derived from the statements above; for example:

$$\begin{aligned} & \mathtt{contains(container\text{-}m,\ node\text{-}A)}, \\ & \mathtt{contains(node\text{-}A,\ node\text{-}B)} \\ \Rightarrow\ & \mathtt{contains(container\text{-}m,\ node\text{-}B)}. \end{aligned} \tag{6.9}$$

`connects`: a `Connects` relationship binds two or more nodes together, via an hyperedge (i.e. a `Connector`) attached to (an internal or external dock on) `Ports` on these `Nodes`. So, statements of the following type are semantically valid:

$$\begin{aligned} & \mathtt{connects}(\mathtt{connct\text{-}i}, \mathtt{port\text{-}s.edock}), \\ & \mathtt{connects}(\mathtt{connct\text{-}i}, \mathtt{port\text{-}u.idock}). \end{aligned} \tag{6.10}$$

6.2.5 Composition

An extra keyword is introduced to indicate that all primitives *in* NPC4 can be compositions in themselves:

$$\texttt{composite} = \{\texttt{node,port,connector,composite}\}.$$

The recursion in this definition reflects the *hierarchical* property of containment in a natural way.

Secondly, the composition with *other, external* DSLs is realised via the following fundamental *design choice*, motivated by the proven way that, for example, XML-based meta-models such as XHTML, SVG or JSON use: each primitive in a model *must* have the following meta data "property tags", that explicitly indicate in which knowledge context (that is, using which meta models) they have to be interpreted:

- `instance_UID`: a *Unique IDentifier* of any instantiation of the primitive concept;
- `model_UID`: a unique pointer to the model that contains the definition of the semantics of the primitive;
- `meta_model_UID`: a unique pointer to the meta-model that describes the language in which the primitive's model is written;
- `name`: a *string* that is only meant to increase human readability.

This generic property meta data allows to compose structural model information with domain knowledge by letting each primitive in a composite domain model *refer* to (only) the structural model that it `conforms-to` [19]; such a composition-by-referencing is a key property of a language to allow for composability.

Finally, since *NPC4* is a language for structural composition, it deserves a separate keyword `compose` to refer to one or more of its possible composition relationships, namely `contains` and `connects`:

$$\texttt{compose} = \{\texttt{has-a}, \texttt{contains}, \texttt{connects}\}.$$

The motivation for the *explicitness* design driver is that (i) *each* of the language primitives can be given its own properties and, more importantly, its own extensions, independently of the others, (ii) it facilitates *automatic reasoning*[4]

[4]This motivation comes from the objective to make the formal models useful not just to human system developers, at design time, but also to *robots themselves*, at *run-time*!

about a given model because all information is in the keywords (and, hence, none is hidden implicitly in the syntax), and (iii) it facilitates *automatic transformation* of the same semantic information between different formal representations. Such *model-to-model* transformations become steadily more relevant in robotics because applications become more complex, and hence lots of different components and knowledge have to be integrated. Trying to do that with one big modelling language becomes increasingly inflexible, because it will be impossible to avoid (partial) overlaps of the many DSLs that robotics applications will eventually have to use in an integrated way.

6.3 *NPC4* Language Constraints

The proposed *NPC4* language not only introduces *primitives* and *relationships*, but also *constraints* to guarantee both syntactic and semantic correctness. In this section these constraints will be discussed.

6.3.1 Constraints for Structural Well-formedness

Some constraints must be satisfied by composition relationships in a graph model to make sure that the model is **well-formed**.

There must be no "floating" ports:

$$\forall \texttt{P} \in \{\texttt{port}\}, \exists ! \texttt{N} \in \{\texttt{node}\} : \texttt{has-a(N,P)}. \qquad (6.11)$$

The reason is that ports get their semantic meaning only from giving access to the behaviour that is contained in the node they belong to, so: without a node, a port has no meaning.

`contains` relationships on `Nodes` must result in a *containment tree*.[5]
A `Node` can contain other `Nodes`, but it must not contain itself. Furthermore, each node has one and only one "direct parent node" in a containment relationship. The reason for this constraint is as follows: since nodes are meant to represent behaviour, and since the containment hierarchy is meant to allow levels of abstraction in a system model, it makes no sense if two nodes that are separated at a higher level of modelling would contain the same behaviour node at a more detailed model level. In other words, behaviour cannot be shared by two nodes with different identity.

As an example, Figure 6.4 visualizes the node containment tree of Figure 6.3. The *node* containment tree is unique for each *hierarchical hypergraph* and plays

[5]Strictly speaking, a *forest*, that is a collection of disjointed trees.

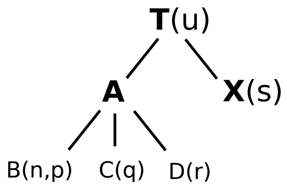


Figure 6.4: The containment tree of the nodes in Figure 6.3. Each node carries its ports as arguments, since this information is required to check the well-formedness of `Connectors`.

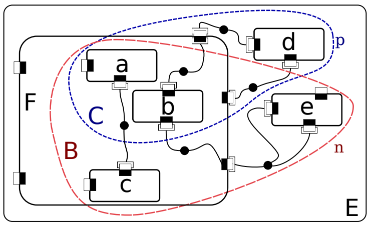


Figure 6.5: An example of a hierarchical composition in which *containment* does not follow a strict *tree* hierarchy *for containers*: the containers "p" (small blue dashes) and "n" (long red dashes) have some internal `Nodes` in common, with each other and with `Node` "A"; the containers "p" and "n" do not have ports themselves, in contrast to the `Node` "A". The *nodes* and their *connectors* do satisfy the *node containment tree* constraint.

a relevant role on determining the validity of a `connects` relationship, as it will be discussed in the following paragraphs.

`contains` relationships of *containers* must result in a *directed acyclic graph*.
That means that a container (or a node) can have multiple "parent containers", and containers can overlap, but cannot contain themselves. This constraint is weaker than that for nodes, since containers are meant to represent knowledge, and knowledge can be shared indefinitely between nodes with different identities. An example is shown in Figure 6.5, where `Containers` `n` and `p` overlap.

A `Connector` `connects` `Ports` on a joint containment tree.
The role of a `Port` is to provide a specific *view* on the `Node` that belongs to. In other words, the effect of the `Port` is to split the containment tree in two sub-trees, considering the `Node` as origin. The `Port`'s *internal dock* selects the "downward" subtree from that `Node`, while the external dock selects the "upward" subtree. Establishing a connection with a specific *dock* means to bound the relationship in the selected subtree, despite the other. For example,

if a connector attaches to an internal dock of a port on a `Node`, all its other attachments must be to external port docks of `Node`s that are contained in the given `Node`, or to other internal port docks of the same `Node`.

For the sake of clarity, Figure 6.6 shows different model examples. The procedure to check this constraint is straightforward when starting from the `Node` containment tree: each of the ports involved in a connector prunes the `Node` containment tree in an downward and upward subtree depending on whether the `Connector` attaches to the `Port`'s internal or external dock, and the tree that remains after considering all involved `Port`s must still be *connected.*

The semantic meaning of this structural constraint is explained by observing an *ill-formed* example reported in Figure 6.6. For instance, the `Connector j` in *model (5)* is semantically not correct, since it relates the node `Node E` with the whole `Node D`, but also with a `Node D` *internal* (`Node A`). Of course the `Node E` can have multiple kind of relationship with the `D Node`, but these are necessarily different relationships, as showed in the *well-formed* model *(*3). Different semantic meaning is represented by the `Connectors (j,k)` in models *(*2) and *(*3). In the former, `Node E` is in relationship with `D`, exposing a specific *view* on it (`Nodes A` and `C`). That is, the coupling `E-A` and `E-C` is indirect, since it considers explicitly the containment boundary `D`. In model *3*, `Connector j` relates directly `Node E` with `A` and `C`, while `Connector k` is a completely unrelated relationship with respect to `Connector j`.

A *well-formed* `Connector` is contained in the *Lowest Common Ancestor* (LCA) of the `Nodes` involved.
Considering the example in Figure 6.3, a statement of the following type is semantically correct:

$$\texttt{contains(node-A,connector-j)}, \tag{6.12}$$

since `node-A` is *LCA* of `Nodes A`, `B` and `C`. This property is a consequence of a *well-formed* `Connector`, and it is not necessarily used to explicitly define a model. In fact, a `Connector` instance is already fully defined by a list of `connects` relationship that involves that `Connector`. However, adding this extra information in a *NPC4* model can be useful as "checksum" during the validation phase. Finally, this `Connector` property helps the rendering of the hierarchical hypergraph layout.

As corollary, that implies that *every* graph model must have *at least one root Node*

$$\forall \texttt{C} \in \{\texttt{connector}\}, \exists \texttt{N} \in \{\texttt{node}\} : \texttt{contains(N,C)}.$$

The reason is that everything inside that root `Node` must have an identified context.[6]

When describing the design decisions behind a formal modelling language, it is not only important to identify and motivate the constraints that compositions in the language must satisfy, but also why some constraints have *not* been introduced in the language. In this work, the following "non-constraint" is one of the fundamental *design choices*: the `contains` and `connects` relationships are *maximally decoupled*, in that one does not depend on the other. For example, even though `Nodes` "X" and "B" live at two different levels of the containment hierarchy, the connector "i" can still connect both (through a port).

Figure 6.5 shows an example in which a connector is crossing a containment boundary, or, in other words, connectors can leave a container without the *explicit* need for a port on that container.

While the decoupling is maximal, it is *not total*: connectors must take the `Node` containment hierarchy into account to some extent, that is, as described by the last constraint above.

In summary, *NPC4* does *not* introduce the (most often implicit!) constraint of interpreting a *containment boundary* also as a *connection boundary*, since this should only be decided (explicitly!) when domain specific semantics is being added to the domain-independent semantics provided by *NPC4*. Another adopted "non constraint" design choice regards the *direction* over a `connects` relationship: no explicit direction is assumed, thus all the connections are *bi-directional*. The direction is a property which belongs to the *behavioral* model, and not to the *structural* one: the constraint will be added in the specific domain of the *meta-model*. The *SDG* model is an example, and further details are in Section 6.5.

[6]This context need not be *unique*, since others can be added by composition.

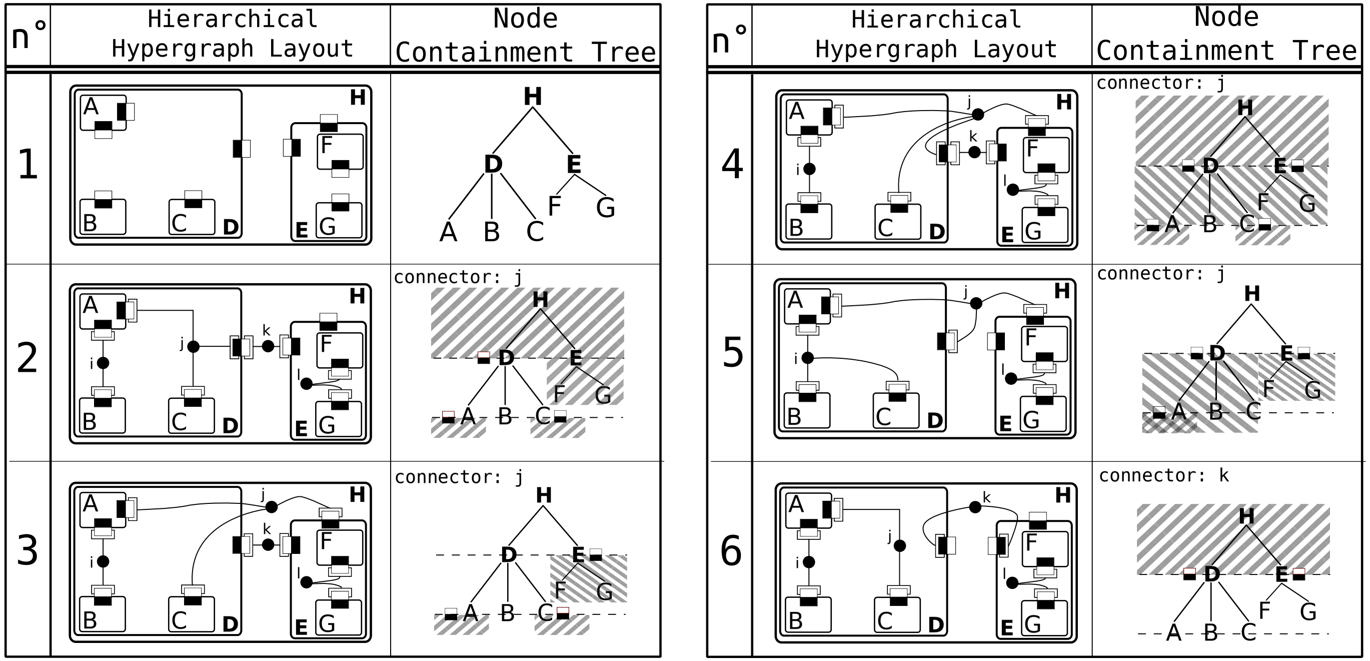


Figure 6.6: Different abstract examples of structural models defined with *NPC4*. Both graphical layout and relative containment tree have been reported. All the examples are based on the first model, which defines `contains` relationships only. The models differs on the `connects` relationship, and the containment tree is not affected by these changes. Examples (2) and (3) show well-formed models. In the associated containment tree, the `Connector` *j* is considered and the `Ports` involved in the relationship are indicated. In both cases, the resulting sub-tree obtained by pruning portions discriminated by the `Ports` is valid. The models (4), (5) and (6) are *ill-formed* because of the presence of a wrong `Connector`. In detail, in Model (4) the relation `connects(j,port-d.idock)` invalidates connections with `node-d` internals, thus the connection is not feasible. In example (5), `port-D.edock` excludes possible connections with `Nodes` `A`,`B` and `C`; since a `Port` in `A` is connected, the `Connector j` is not correct. The latter case (6) shows an intuitive case of connecting two `Nodes` through two wrong `docks` (`Connector k`).

6.3.2 Constraints Formalisation

Section 6.2.3 introduced the *primitives* of the *NPC4* language, and the `contains` and `connects` *relationships* that can exist between these primitives. However, not all relationships that can be formed syntactically also have semantic meaning. This section describes some *constraints* already discussed in the previous section, but striving for formal completeness, by adding some obvious constraint relationships to the core semantic explained above.

Note that no `connects` relationships appear anywhere in the constraints on the `contains` relationships, and the other way around, which reflects the above-mentioned *orthogonality* of both relationships. Of course, when application developers add behaviour to a structural model of their system, they may introduce *extra* structural constraints, even between `has-a`, `connects` and `contains` relationships.

Constraints on primitives. The `UID` of every primitive must be unique:

$$\forall(\texttt{X,Y}) \in \{\texttt{node}, \texttt{port}, \texttt{connector},$$

$$\texttt{contains}, \texttt{connects}\},$$

$$\texttt{X.UID} = \texttt{Y.UID} \Rightarrow \texttt{X} = \texttt{Y}.$$

Of course, these constraints hold for all three UIDs in the meta data of each *NPC4* primitive.

Constraints on `has-a`. As mentioned in Section 6.2.4, a `Port` can be floating during construction time, but a model having a port that is not `part-of` a `Node` is an *ill-formed* model. Furthermore, a `Port` *must* be `part-of` one and only one `Node`, that is:

$$\forall \texttt{P} \in \texttt{\{port\}}, \forall(\texttt{N1,N2}) \in \texttt{\{node\}},$$

$$\texttt{has-a(N1,P)}, \texttt{has-a(N2,P)} \Rightarrow \texttt{N1} = \texttt{N2}.$$

The previous statements affects other relationships too, as it will be shown in the next paragraph.

Constraints on `connects`. The constraints in this section realise the *well-formedness* of the connection relationships, that is, about which kind of structural interconnections are possible. Recalling from Section 6.2.4, the `Port` has exactly two `docks`, one *internal*, and one *external*. Each `dock` is constrained to have only one `Connector` attached, that is:

$$\begin{aligned} & \forall(\texttt{C1},\texttt{C2}) \in \texttt{\{connector\}}, \forall \texttt{P} \in \texttt{\{port\}} : \\ & \texttt{connects(C1,P.idock)}, \\ & \texttt{connects(C2,P.idock)} \\ \Rightarrow\ & \texttt{C1} = \texttt{C2} \end{aligned}$$

$$\begin{aligned} & \forall(\texttt{C1},\texttt{C2}) \in \texttt{\{connector\}}, \forall \texttt{P} \in \texttt{\{port\}} : \\ & \texttt{connects(C1,P.edock)}, \\ & \texttt{connects(C2,P.edock)} \\ \Rightarrow\ & \texttt{C1} = \texttt{C2} \end{aligned}$$

Furthermore, the *well-formedness* of the `Connector` (discussed in Section 6.3.1) can be formally expressed as follows:

- given `C` is the `Connector` to be validated;
- given the sets of internal and external `Ports`, p_{ci} and p_{ce}, defined as:

$$p_{ci} \triangleq \{p \in \texttt{\{port\}} \mid \texttt{connects(C,}p\texttt{.idock)}\}$$

$$p_{ce} \triangleq \{p \in \texttt{\{port\}} \mid \texttt{connects(C,}p\texttt{.edock)}\}$$

- then, $\forall p_i \in p_{ci}$, $N_i \in$ `{node}` s.t. `has-a(`N_p, p_i`)` holds, $\forall p_j \in \{p_{ci}\} - p_i$, `C` `Connector` is valid if `contains(`N_p, p_j`)` holds, and the following condition holds
- $\forall p_e \in p_{ce}$, $N_i \in$ `{node}` s.t. `has-a(`N_p, p_e`)` holds, $\forall p_j \in \{p_{ce}\} - p_e$, `C` `Connector` is valid if `contains(`N_p, p_j`)` does **not** hold.

Constraints on `contains`. The constraints in this paragraph realise the *well-formedness* of the containment relationships of `Nodes`, that is, about which kind of hierarchies, or “composites” are possible.

First, the fact that *every* primitive *can* be a `composite` in itself is expressed:

$$
\begin{aligned}
&\texttt{composite} = \{\texttt{node}, \texttt{port}, \texttt{connector}, \texttt{composite}\},\\
&\quad \forall \texttt{C} \in \{\texttt{composite}\} :\\
&\quad\quad \exists \texttt{n} \in \texttt{\{node\}} \vee \exists \texttt{c} \in \texttt{\{connector\}}\\
&\quad\quad \vee \exists \texttt{d} \in \{\texttt{composite}\} :\\
&\quad\quad\quad \texttt{contains(C,n)} \vee \texttt{contains(C,c)}\\
&\quad\quad\quad \vee \texttt{contains(C,d)}.
\end{aligned}
$$

Every `contains` relationship can only be defined on *existing* `Node`s and containers:

$$
\begin{aligned}
&\forall \texttt{c} \in \{\texttt{contains}\},\\
&\quad \exists \texttt{(X,Y)} \in \{\texttt{node,container}\} :\\
&\quad\quad \texttt{c(X,Y)}.
\end{aligned}
$$

And finally, there *always* exists at least one `Node` at the top of a `contains` hierarchy:

$$
\begin{aligned}
&\forall \texttt{X} \in \{\texttt{node,connector,composite}\},\\
&\quad \exists \texttt{T} \in \{\texttt{Node}\} : \texttt{contains(T,X)} \vee \texttt{T=X}.
\end{aligned}
$$

This latter constraint is a *very strong* one, that is imposed for one and only one reason: every structural model should have an explicitly identified *context*. In other words, the meta data of the top `Node` must be made rich enough to understand the semantics of *everything* it `contains`, even when the model is deployed in a running system. There can be *more than one context* for each composition, which is in agreement with the design objective of composability: several context containers can be put around any existing model, and/or a composite can *conform to* more than one meta-model. The top `Node` need not have any `Port` attached to it, so that it reduces to just a *container of meta data*.

6.4 Modelling with *NPC4*

This section briefly discusses some structural features of the proposed solution.

6.4.1 Structure for Supporting Software

Many domain models use only traditional graphs, with `Nodes` and edges, while the proposed hierarchical hypergraph model splits the "edge" primitive in two new first-class primitives: "port" and "connector". The motivation for this choice is to allow not only more precise *domain* semantics *if needed*, but also a more flexible *infrastructure to support* a domain model with *software*. For example, by using ports to log and visualise data exchange between `Nodes`, or to count the number of interactions (statically as well as during run-time), or to make graphical development tools in which selections have to be made on which internal behaviour of `Nodes` to connect to, and so on.

Recall also the other motivation of this work with respect to *hierarchical composition*: at a certain level of abstraction of a system model, a port might be a completely passive part of a system model, that is, without behaviour of its own, while more behaviour appears when going to a deeper level of abstraction in the system part represented by that port. A typical example is communication: two `Nodes` connected with communication middleware send and receive data through socket ports, at the application layer, but when going inside such a socket at the level of the operating system, lots of activity becomes visible: packet composition, encoding, time stamping, and so on. Much of that activity is "infrastructure" code for the higher level of abstraction, but the suggested approach allows to connect all these things together, over different levels of abstraction.

The third software-centric motivation for the presented model pertains to the introduction of the *container* primitive: it *carries no behaviour*, but is used to model *information influence* of "higher" contexts[7] on "lower" `Nodes`, ports and connectors. More precisely, the container model primitive is needed *to store meta data*, such as: unique identifiers; references to the modelling languages in which the `Nodes`, ports or connectors inside a container are expressed; references to ontologies that encode the semantic meaning of the model (hence indicating, among other things, which configuration values to use for all model parameters); version numbers; etc. One particularly useful case is to introduce containers to store the *composition model* of the sub-system that is embedded within its internal context.

[7]Or scope, namespace, domain, or any other terminology has been used to represent this container concept.

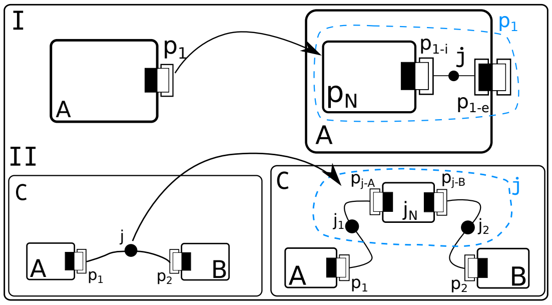


Figure 6.7: Examples of possible model-to-model transformation to describe a deeper level of abstraction. The first example (I-right) shows a solution to model behaviour on a `Port` primitive (I-left): `Port` p_1 is expanded in a `Node` p_N contained in `A` (original owner of p_1), while an internal `Connector` establishes a *view* over the same `Node`. The containment tree changes only internally to `A`, thus the change is compliant with the original model. Not necessarily but useful, p_1 refers to a `Container` in the transformed model, such that the original semantic information is preserved (and it is possible to retrieve the original model). The second example (`II`) shows a similar case, but considering the `Connector j` as target of the transformation.

6.4.2 Behaviour on Deeper *Levels of Abstractions*

In the proposed structural *meta-model*, the *hierarchy* concept is applied to `Node` and `Container` primitives only. Allowing `Ports` and `Connectors` being hierarchies on themselves would violate the design choice that only `Nodes` carry behaviour.

However, in practical cases `Ports` and `Connectors` may manifest behaviour, if a *deeper level of abstraction* is considered. A concrete example arises in the attempt of model a software system involving two computers: what was first a simple shared data structure (*i.e.*, a "`Connector`") in the centralized version now becomes a full set of cooperating "middleware" software components in itself (*i.e.*, a composition of `Nodes`, `Connectors` and `Ports`). In short, modelling the distribution explicitly boils down in introducing a deeper levels of abstraction.

In such cases, it is possible to apply a systematic *model-to-model* transformation to obtain an alternative model, as a *composition* of the original *NPC4* model and a separate *NPC4* model of the `Port` (or `Connector`) internals. Figure 6.7 illustrates two examples which expands `Port` and `Connector` respectively. In both cases `Port` and `Connector` have been modelled as a simple `Node`, which already enables the *hierarchy* feature. Furthermore, a `Container` may be added to preserve the knowledge over the original model. As a remark, this property is offered by the *composability* feature of *NPC4*, considered as one of the primary design drivers of the language.

In conclusion, modeling a deeper level of abstraction is always possible, but the described structural models differs.

6.5 *NPC4*, *SDG* and *uSDL*

This section provides additional insights on the hierarchical hypergraph structure adopted in the *SDG* model (Chapter 2). In fact, the *SDG* model *conforms-to* the *NPC4* DSL by adding domain specific constraints on the top of *NPC4*. This conformity is reflected in the mapping between the *NPC4* primitives and the *uSDL* primitives resumed in Table 6.3; further details are provided below:

1. a `Skill` s is represented as a node in the graph, thus its declaration in *uSDL* corresponds to a `Node` declaration in *NPC4*; in addition, a `Skill` declaration deploys a *NPC4* `Port` hosted by the node itself, representing the intended effect of the skill, `eff`(s);

2. a `Condition` in *uSDL* creates a `Port` in *NPC4*; like the `Port`, `Conditions` are not *"floating"*, but they belong to a `Skill` (*NPC4* `Node`), thus an extra `has-a` relationship must be defined. This structural constraint influences the well-formedness of a *SDG* model;

3. a dependency or logical constraint in the *SDG* model is represented as a `Connector` in *NPC4*, and such a `Connector` can host multiple dependencies;

4. the primitive `contains` does not change between *NPC4* and *uSDL*, since it represents the same hierarchical tree between nodes;

5. *NPC4* `Containers` are structural primitives used to represent **run-time** information over a *SDG*, fundamental for a JIT approach; concrete examples are the "loop unrolling" and the "branch predication" discussed in Chapter 5 (Section 5.3.1 and Section 5.3.2);

6. the primitives `toStart`, `latches` and `continuesIf` in *uSDL* define a relationship between a `Skill` and a `Condition`, thus they structurally `connects` a `Node` to a `Connector`. In detail, such a connection links a `Port` that is hosted by the `Node`, thus an additional `Port` is deployed in that `Node` (`has-a` primitive in *NPC4*). In a well-formed *uSDL* model, the origin of the `Condition` is also known, thus a second `connects` relationship is applied between the *NPC4* `Connector` that represents the logical constraint and

	uSDL		*NPC4*
1	`Skill`	$\longrightarrow$	`Node (has-a)`
2	`Condition`	$\longrightarrow$	`Port`
3	**Dependency**	$\longrightarrow$	`Connector`
4	`contains`	$\longrightarrow$	`contains`
5	**Run-time info**	$\longrightarrow$	`Container`
6	`toStart`, `latches`, `continuesIf`	$\longrightarrow$	`connects (has-a)`
7	`is-effect-of`, `is-side-effect-of`	$\longrightarrow$	`has-a`

Table 6.3: *conforms-to* relationship between primitives and relationships in *uSDL* and *NPC4*.

the *NPC4* `Port` that represents the "source" of the *uSDL* `Condition`, whether it belongs to a *controlled* `Skill` or not;

7. the relationships `is-effect-of` and `is-side-effect-of` in *uSDL* is equivalent to the relationship `has-a` in *NPC4*; both declare a relationship between a `Node` (`Skill`) and a `Port` (intended effect or any other side-effects).

The additional **domain-dependent constraints** that the *SDG* model imposes over a generic *NPC4* model are:

- **directivity:** `Node-Connector` connections (through `Ports`) are unidirectional, and graphically indicated with an arrow in the examples of this dissertation; the arrow points to the `Node` subject to the constraint, the tail to the source of the conditions involved in the relationship;
- **acyclicity:** the *SDG* model is acyclic, and no loops are allowed (but a symbolic plan with loops can be unrolled, see Section 5.3.1).

Finally, the *behaviour* attached to a *SDG* model is directed to its interpretation, that is, the *SDG*-Executive.

6.5.1 Example

The above-mentioned mapping between *uSDL* and *NPC4* DSLs may be overwhelming without some familiarity to both languages; for the sake of clarity, Table 6.4 reports an example of an *uSDL* model and its structural representation in *NPC4*. As expected, the *uSDL* is less verbose than its structural *NPC4* model, since the domain adds syntactic sugar on the overall description. For instance, each `Skill` comes with an intended effect, represented as a *NPC4* `Port` that belongs to the `Skill`.

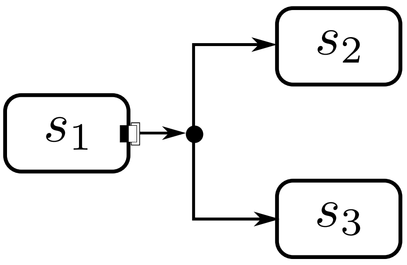
```
Skill: s1, s2, s3

toStart(s2,eff(s1))
toStart(s3,eff(s1))
```

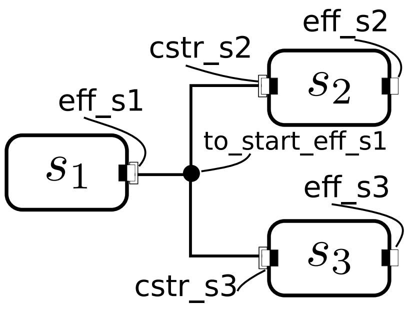
```
Node: s1, s2, s3
Port: eff_s1, eff_s2, eff_s3,
  cstr_s2, cstr_s3
Connector: to_start_eff_s1

has-a(s1, eff_s1)
has-a(s2, eff_s2)
has-a(s3, eff_s3)
has-a(s2, cstr_s2)
has-a(s3, cstr_s3)
connects(to_start_eff_s1, eff_s1.edock)
connects(to_start_eff_s1, cstr_s2.edock)
connects(to_start_eff_s1, cstr_s3.edock)
```

Table 6.4: Example of an *uSDL* model (on the left) and its structural *NPC4* model (on the right).



(a) Graphical representation of the *uSDL* model in Table 6.4.



(b) Graphical representation of the *NPC4* model in Table 6.4.

Figure 6.8: Structural *NPC4* view (Figure 6.8b) of the *SDG* model in Figure 6.8a.

Figure 6.8 depicts both *SDG* and *NPC4* models: the directivity is a clear domain-dependent addition, which is not represented in the structural view.

However, the *NPC4* model contains all the structural elements *explicitly*, *e.g.*, `Ports` representation of the intended effects `eff`(s_2) and `eff`(s_3), which are not used in the *SDG* model.

6.6 Conclusions

What is the minimal set of primitives and relationships, to cover all use cases of structural composition in robotics applications?

This was the main research question addressed in this chapter, motivated by the drive to realise a step change in the reuse of "infrastucture code". Several robotic frameworks have been developed in the last years, and all of them have quite overlapping needs with respect to the structural composition of the functional primitives they offer, yet no common designs or models are shared, let alone code. This chapter advocates the use of the *NPC4* language, as *the* meta-model to represent *port-based composition*, for both *interconnection of behaviour* and *containment of knowledge*, and in a domain-independent way.

The minimal set of *primitives* adopted are common elements: `nodes`, `ports` and `connectors` (or semantically equivalent concepts) have been used in several contexts, in one form or another. The real challenge was to identify the minimal set of *constraints* that govern all structural compositions: the lesson learned is that developers tend to be not very aware of such constraints, and the more expert one is in a certain domain, the more obvious and implicit such constraints appear.

The objectives behind this work are: (i) to separate strictly the structural and behavioural aspects, and (ii) to make *all* structural relationships *explicit* in a formal language, based on *hierarchical hypergraphs*. The benefits of having a common structure are manifold, such as common tools for storing, querying and composing heterogeneous systems, as well as easily create new funcionalities or DSLs based on the graph structure.

The behaviour attached to the structural primitives always depends on the specific *context* in which various pieces of the knowledge integrated in the system are valid or not. Hence, it is important to have an explicit computer-readable representation of the *structural knowledge contexts* in which a system is contained; most often, there are many overlapping contexts active at the same time. Hence, the hierarchical hypergraph meta-meta-model is highly relevant to make the step from traditional engineering systems to *knowledge-aware* engineering systems, that is, systems that can *use the knowledge themselves at run-time*.

In the above-mentioned context, the aspect of *composability* of structural models is an important design focus; *NPC4* advocates that extra “features” (such as behaviour or visualisation) should not be added “by inheritance” (that is, by adding attributes or properties to already existing primitives), but “by composition”, that is, a *new* DSL is made, that imports already existing DSLs and adds *only the new* relationships and/or properties as *first-class* and *explicit* language primitives. An example is the *SDG* model of Chapter 2 and its language *uSDL*, which adds to the structural model acyclicity and directivity constraints.

Although presented in a robotics context, nothing in *NPC4* depends on this specific robotics domain. *NPC4* can also serve the goals of related to other application domains such as the *Internet of Things*. However, the advantages of the *NPC4* meta-model pay off most in robotics, because of (i) the large demand for *knowledge-aware* systems, (ii) the online efficiency and (re)configuration flexibility of such robotics systems, and (iii) their need for the online reasoning about (and eventually the online adaptation of) their own structural architectures.

Finally, this work suggests the *NPC4* language for adoption as an *application-neutral standard*, since standardizing the structural part of components, knowledge, or systems, is a long-overdue step towards higher efficiency and reuse in robotics system modelling design, and in the development of reusable tooling and (meta) algorithms.

Chapter 7

Conclusion

This chapter concludes this dissertation, summarising the contributions and the results obtained with respect to the problem statement of Chapter 1. Moreover, a critical discussion on the improvements with respect to the state-of-the-art is provided in Section 7.2, while concrete suggestions for future work are illustrated in Section 7.3.

7.1 Contributions

As its main contribution, **this dissertation shows how to bridge symbolic plans to executable constraint-based task specifications by means of offline skill programming, and online skill scheduling and adaptation**. This work suggests a solution to the representation problem of actions in both **discrete** and **continuous** domains, advocating a **declarative approach in place of procedural specifications**, and **postponing** the grounding of symbolic actions until it is needed.

This thesis proposes a set of formal models from which the following *Domain Specific Languages* are derived:

- the *micro Skill Dependency Language* (*uSDL*) that includes three declarative rules (*i.e.* `toStart`, `continuesIf` and `latches`) sufficient to express dependencies over the skill execution;
- a geometric-based task specification that expresses robotic tasks in a constraint-based fashion by means of relationships between entities;

- the *NPC4* language to describe the structure of hierarchical hypergraphs.

The knowledge shows up to any level and in many forms, providing those **context-dependent** information necessary to adapt the execution of the skill. This work denotes the causes that turn the context of the execution being variable (*i.e.*, object(s) involved in the task, robot capabilities, purpose of the action and environment), and exposing those in the specification. To exploit this knowledge, a *skill prototype* describes the grounding of a task specification that suits to some contexts; such a model is fed **Just-in-Time** with additional knowledge, and the outcome is an executable *skill instance* adapted to a specific context.

Furthermore, this work promotes the *Skill Dependency Graph* as a good formal model to represent the dependencies between motion skills and logical conditions that influence the skill execution. These conditions that abstract the state of the world reside in the discrete domain, and they must be mapped to the monitoring of continuous quantities. Moreover, the choice of using *graphs* to describe dependencies is already applied to other domains, and this motivates the modelling effort to define their structure, aiming to a standardisation of such an approach.

The solutions presented can be adopted as a whole, but also individually: the geometric-based task specification is employed in other coordination models such as Finite State Machines; the skill model employed in the *SDG* is back compatible with skill models of other frameworks, where the activation of the skill is atomic or implemented as a functional delegation; the *Just-in-Time* approach holds for other workflows; *hierarchical hypergraphs* and the *NPC4* cover structural models in other domains. However, the main benefits are visible considering the overall integration that define a complete plan executive. As a matter of fact, such a framework combines an *object-centric* DSL based on geometric expressions and a declarative *action-centric* DSL (*i.e.*, the *uSDL*), providing a synergy among these approaches.

7.1.1 Maturity of the Models and Software Support

This work proposes formal models and DSLs, but according to the Gödel's incompleteness theorems [127], it is not possible to demonstrate that those models are complete and consistent, neither the validity of the axioms in which they are founded. However, it is possible to argue about the maturity of the models, expressed as the stability achieved after a certain number of design iterations and changes. In the same vein, it is possible to report about the maturity of the software developed in this thesis to support the necessary mechanisms that "activate" the models. A short overview is discussed below.

Skill Dependency Graph and *uSDL*
The Skill Dependency Graph has undergone multiple design iterations and its formalisation is rather stable; the *SDG* structural model and the formulation of the dependencies (activation and invariant) form a *meta-meta-model* to which the *uSDL* (meta-model) *conforms to.* Since the executive (*SDG*-E) is implemented on the basis of the propositional logic described by such a meta-meta-model, the software is ready to support other languages that implement changes to the behaviours represented in the skill model, as well as changes to the transitional model (*SESD*). However, both skill model and *uSDL* suffice to cover several examples reported in Chapter 4.

Geometric-based Task Specification (Chapter 3)
This DSL has three years of development within the context of the *RoboHow.cog* [128] already, and numerous design iterations[1] shaped this meta model at its current status. Originally, the task specification was implemented as *internal DSL* in the Lua [70] language, and only recently is consolidated as a JSON format description, increasing the chances of diffusion and adoption. The software that converts a task specification into a constraint-optimisation problem is rather stable, as well as the cases that the language covers.

Just-in-Time (Chapter 5)
The skill life cycle presented in Chapter 5 allows to implement a Just-in-Time strategy for the online refinement of a skill prototype to a concrete executable instance. Such a model is implemented, together with the metric that determines whether to refine the skill or not, in the *SDG*-E that executes the examples of Chapter 4. However, this result is recent, and the software support is still at prototype level (alpha phase). Further development is not only needed for the software implementation of the Just-in-Time strategy in itself, but also for investigating to what extent mature compiler technologies (*e.g.*, LLVM[2]) could be used to fully exploit the potential of this approach.

Hierarchical Hypergraphs and *NPC4* (Chapter 6)
Hierarchical hypergraphs are well-known in mathematics as an extension to traditional graphs. The contribution is on their usage as formal models to separate structure and behaviour, as well as on the introduction of the *NPC4* language and its role to represent both novel and existing graph-based algorithms. The design of *NPC4* has several years of maturity, and the contribution of many co-authors, also within other projects such as

[1]The author is grateful to Prof. Michael Beetz from University of Bremen and Prof. Abderrahmane Kheddar from CNRS, University of Montpellier for their precious suggestions.
[2]http://llvm.org

ROSETTA [130] and *BRICS* [20]. At the time of writing, a critical aspect is the limited support of software and tooling to provide those common functionalities that could enlarge the number of adopters.

7.2 Discussion

This section provides a critical discussion about advantages, disadvantages and novelty of the presented research with respect to the state-of-the-art and alternative frameworks that ground and schedule symbolic actions into robot motions. A fair comparison is not trivial, due to the lack of common benchmarks, reproducibility of the results, and the overall complexity those frameworks. Therefore, this evaluation is *qualitative*, based on the improvements that this dissertation introduces.

As a first remark, this work is distinguished among the state-of-the-art for the focus on explicit formal models, while other works focus on functionalities only; this approach permits a comparison with respect to prior work, separating *what* the different frameworks provide, and *how* a certain functionality is delivered.

7.2.1 The Stack of Tasks Framework, SoT

The *Stack of Tasks* (SoT) [96, 97] provides a stack-based scheduler to insert (and remove) constraints in the numerical solver responsible for computing the control action. The SoT architecture is composed of three layers that rule the stack of tasks: the first two layers handle conflicting constraints with respect to the so-called *environmental* constraints (*e.g*, joint limits and collision avoidance); the third layer, namely *look-ahead controller*, prevents the convergence of the solution to a local minimum or a dead-lock situation. These layers represent a concrete mechanism that conforms to the semantics expressed by the DSL proposed in Chapter 3. In detail, the constraints managed by the first two layers are of type *safety* (*e.g*, joint limits) or *primary* (*e.g*, collision avoidance), while the remaining are of type *primary* or *auxiliary*, depending on the application. In addition, the collision avoidance may be modelled as a set of *safety* constraints as well, but this choice must be application dependent: in fact, it is necessary to differentiate when a collision is desired, when it is forbidden, and when it is not relevant for the application. Instead, the SoT imposes the role of the collision avoidance, that is a strong assumption; while the proposed geometric-based DSL permits to express such a role very flexibly in the task specification, the SoT hides one explicit choice of that role in the implementation. Since the SoT software provides solvers for a subset of the task specifications that are covered by the proposed geometric-based DSL, demonstrations integrating both are realised in the context of the *EU FP7 RoboHow.cog* project [128].

A SoT application is realised as a plain sequence of pre-defined constraints, with some degrees of configurability over the formulation of the COP, namely the insertion and removal of some constraints. The work of Keith *et al.* [78, 79] extends this approach formulating a scheduler based on a Constraint Satisfaction Problem (CSP). Given a set of tasks, the scheduler is responsible for computing their execution order, considering temporal constraints between task pairs. In this context, the temporal constraints are derived from a subset of Allen's Interval Algebra [4], since each task is modelled as a time interval that represents the duration of the task execution. The outcome is a *task temporal network* that optimises the execution of the whole plan. This solution is technically sound, and it is widely used in planning literature [56, 111], as well as in robotics [107]. However, the expressivity of this approach is rather limited, since a temporal constraint defines an implicit relationship between a task execution and the intended *effect* produced by the execution of another task: this means that *i)* the execution of a task cannot be constrained to an effect caused by another agent (*e.g.*, a human pressing a button), and *ii)* the plan can represent only nominal situations. In addition, the optimization process proposed in [78] requires the prior knowledge of the *exact* duration of each task execution. In practice, it is not straightforward to compute those timing, even by simulating the *nominal* behaviour beforehand, which is also computational expensive. Resuming, the scheduling optimisation suggested in [78, 79] resides in the planning domain, and it is limited in expressivity and reactiveness. The *SDG* model proposed in Chapter 2, together with the *Just-in-Time* approach (Chapter 5), solves these limitations by modelling *explicitly* those logical constraints and alternative behaviours that permit to adapt and optimise the plan execution on-line.

7.2.2 The KU Leuven Approach: iTaSC and eTaSL

iTaSC

The *instantaneous Task Specification using Constraints* (iTaSC) framework [36] provides a solution to integrate both instantaneous task specification and estimation of geometric uncertainties, but it does not provide any explicit mechanism to coordinate or sequencing a set of task specifications. Smits *et al.* [151, 150] introduces the concept of skill within the iTaSC framework: the behaviour (*i.e.*, the task) to realise is implemented in a constraint-based fashion, and a discrete FSM (precisely, a State Chart diagram [66]) coordinates which task specification is executed. Strictly speaking, each FSM state represents a COP to be solved, and its execution performs a guarded motion: if a certain condition occurs (success or failure), a switch to another COP is applied. Recently, Vanthienen *et al.* [168] extended such a mechanism, introducing the so-called *Composition Pattern* and a specific DSL for deploying iTaSC applications [168]. However, the synthesis of a skill remains unchanged, since an analogous procedural-based approach is adopted (precisely, the *reduced Finite State Machine*, rFSM [81]).

The shortcomings of such a methodology are the ones that motivated this research, that is: *i)* numerical values are defined offline, *e.g.*, weights on the optimisation function terms and control gains; *ii)* the FSM that coordinates the overall application is statically defined and often hand-written; *iii)* "concurrent execution" (*e.g.*, opening a gripper while approaching to the tray in an open a drawer scenario) is provided by a manual composition of the constraint-based (sub-)task into a single activity description; *iv)* the only logical conditions are those that enable a transition in the FSM model, while no explicit constraints denote when it is legal to aggregate some (sub-)tasks in a single activity, and when it is not; in addition *v)* there is no explicit mechanism to model the reasons why a state in the FSM is considered to be a failure or a non-nominal behaviour. All the above lead to a formulation of a COP which is well-formed only under some design assumptions that hold in a known and controlled environment. Therefore, the claimed re-usability of task specification is limited to a model of the world that is defined offline (*i.e.*, closed-world assumption), and as a consequence, the same plan on a similar context may require to encode another FSM model, as well as the tuning of a numerical configuration.

The work presented in this dissertation addresses to the above-mentioned issues, providing: *i)* a way to schedule and to adapt a skill execution from a set of explicit declarative rules, namely the *uSDL* (Chapter 2), *ii)* the skill model (Chapter 2) offers a structural way to express non-nominal and failure behaviours, *iii)* the skill prototypes (Chapter 4) are composable models that specify a task by symbolic geometric relationships (Chapter 3), re-usable under well-defined contexts, and *iv)* a mechanism that transforms these prototypes into skill instances that are optimised to the context in which they are executed (Chapter 5), hence allowing an online *"opening"* in the model of the world.

In addition, the geometric-based DSL in Chapter 3 defines constraint-based tasks on the basis of non-causal geometric relationships, alternative to the *Virtual Kinematic Chain* (VKC), which are each particular instances of casual representation of the same relationship.

For the sake of comparison accuracy, the iTaSC methodology includes a solution to the estimation of geometric uncertainties, a problem that is not treated in the scope of this work.

eTaSL
An alternative framework is the *expressiongraph-based Task Specification Language* eTaSL [2] that introduces an *Automatic Differentation* mechanism to compute the Jacobian of multiple expressions. On this basis, eTaSL proposes a scripting-based language to define a task specification, then translated to a numerical COP, precisely, a Quadratic Programming (QP) problem. The eTaSL advocates the usage of auxiliary variables, namely the *feature variables*, that specify free movements not necessarily associated to a robot joint or to a physical object. The *feature variables* offer a solution to formulate complex constraints by coupling expressions together, and the composition is directly reflected in the COP formulation.

The geometric-based DSL (Chapter 3) does not support explicitly feature variables, however: *i)* geometric items (Chapter 4) are not necessarily attached to a physical object, but they can also represent virtual objects that describe free movements too; *ii)* the constraint coupling with feature variables is strong, that is, it expresses a well-defined situation, and it is not meant to be combined with other skills. Therefore, the usage of feature variables as suggested in [2] exhibits a symptom of early optimisation, since the task specification can be executed only "as is", and there is no symbolic attachment that allows to reason about feature variables. Nevertheless, feature variables turn out to be useful in those well-defined cases where the task is rather specific, described by a skill developer and it must not be (automatically) combined with others. The *SDG* model supports those cases: the skill is executed "stand-alone", as a strict sequence with respect to previous and subsequent skills. It is the author's suggestion to do not abuse on the usage of the feature variables where it is not necessary.

Another difference between eTaSL and this work resides in the methodology adopted. As a matter of fact, the eTaSL language does not provide a formalism neither on the syntax nor on the grammar, and the specification is strongly coupled with the underlying implementation, that is, it is not a DSL but a module of the Lua [70] scripting language.

The eTaSL framework focuses on the continuous domain of the task specification, and it does not support run-time (re-)configuration of a task. In detail, the language boils down to an offline configuration of sequential tasks ruled by a FSM: the discrete coordination, as well as numerical information must be known prior to execution. In case the latter can be retrieved by external sources, they must be known and the related resources (*e.g.*, communication) must be allocated at configuration time. The proposed Just-in-Time methodology in Chapter 5 aims to improve the above-mentioned limitations.

The implementation of the geometric-based DSL (Chapter 3) and the experiments illustrated in Chapter 4 share the same *expressiongraph* library[3,4] that also realises the eTaSL framework.

7.2.3 Other Solutions in the Planning Domain

This dissertation addresses the problem of bridging the gap between symbolic plans and executable motion descriptions, with a preference on optimisation-based motion control solutions. To the best of the author's knowledge, the works in this direction are few. For example, the research presented in [120, 121] aims to bridge the symbolic planner ICARUS [90] with the *Stanford Whole Body Control* (SWBC) framework: a skill describes a constraint-based task that exposes some parameters configured at run-time, but it is not clear when these parameters feed the skill model. Moreover, the skill activation is driven by a FSM, no online composition is supported, and non-nominal behaviours are not modelled. However, this work shows a solution that interacts with a symbolic planner, while the focus of this dissertation is limited to the execution of symbolic plans.

An alternative methodology to bridge planning and acting consists to integrate the continuous motion planning in the reasoner [47, 74]. In these approaches, the motions are often the result of the execution of pre-computed trajectories, which are in turn generated from more target configurations that are then checked for feasibility and reachability. Generally, the path planning is solved in the joint space, *e.g.*, by a RRT-based method (Rapidly exploring Random Tree), and in case of environmental changes a replanning is necessary. In [18] object geometry is taken into account to reduce the search space of the motion planning. Another approach is to interleave the symbolic planning with geometric backtracking [37], such that the reasoner can infer alternative motion plans in case of failure of a previous execution. An interesting constraint-based solution is provided in [89, 88], that combines both task and motion planning: this reasoner takes into account multiple constraints from the context of the execution, *e.g.*, the geometry of the objects, robot redundancies, and a sequence of the symbolic actions that must be performed. Similar results are obtained in [156]: instead of proposing a novel planning algorithm, it explores the possibility of interleaving classical symbolic planners with traditional motion planner generators.

The above-mentioned solutions reside in the planning domain, bridging symbolic plans and continuous motion planning based on constraints. These approaches differ from the methodology proposed in this dissertation, since the latter focuses

[3]`https://github.com/eaertbel/expressiongraph`

[4]The author is thankful to Erwin Aertbeliën for publicly releasing the expressiongraph library.

on constraint-based motion control: *i)* in the planning, information about the context must be known and evaluated in advance (eager evaluation), while in the JIT approach the need of such a knowledge is delayed as long as possible; this benefit is tangible in case of dynamic environment, in fact *ii)* in case of environmental changes, a replanning is necessary for motion planning solutions (*e.g.*, if an object is moved); instead, constraint-based motion control does not require any update in the task specification for small changes, and the *SDG* model contains all the knowledge to deal with local changes; *iii)* such replanning is also due to the lack of semantic information on a generated path to realise, thus it is not possible to reason about local changes (apart of the geometric backtracking); using explicit constraints to generate the control action allows to reason about the motion itself, since constraints have a semantic meaning; *iv)* the solution proposed in this dissertation allows to specify and to realise *force-based* and *physical interaction* with (moving) objects (including human-in-the-loop situations, not discussed in this dissertation); on contrary, motion planning solutions are limited in that perspective, even when the motion plan is generated considering both kinematic and dynamic constraints on the robot and the manipulated object; lastly, *v)* symbolic planning algorithms are computationally expensive *NP-hard* problems[5], and integrating those with motion planning is quite demanding; on contrary, the task executive (*SDG*-E) proposed in this dissertation runs at the same frequency of the control loop (namely, up to 250 Hz). However, the proposed framework is also affected by few disadvantages, among which *i)* the optimisation problem may suffer of local minima, *ii)* the motion control, and the whole framework provided best fits to local problems, and *iii)* it provides a solution to ground symbolic plans, but not to generate those. Anyhow, a motion planning approach does not exclude an integration with reactive control: they are complementary, and not alternatives.

7.2.4 Preview Coordination

The *SDG* formalism presented in this work has not been the first solution by the author of this dissertation; an early approach is the *Preview Coordination* presented in [139]. The *Preview Coordination* aims to optimise the overall execution time of a robotic application; the core idea is to augment an existing FSM model with run-time information about the likelihood of the future transitions: an extension of the FSM execution model determines the most likely future activities, and it executes those that are compatible with the current activity and robot capabilities.

[5]NP-hard, Non-deterministic Polynomial-time algorithms are those that cannot be solved in a deterministic, polynomial time.

In detail, the employed method converts a regular FSM model to a *Deterministic Probabilistic Finite-State Automata* (DPFA) [172], from which a *preview state list* is computed online. This list contains the likely future activities within a pre-defined preview horizon, and their execution is enabled by a *task activation policy*. Such a policy considers possible conflicts between the candidate future activities and the current one, as well as an activity progress parameter that indicates how close is the current activity to terminate; in this way, an early future activity is inhibited from its execution if the expected benefits from an early execution are not tangible.

The Preview Coordination has not been integrated with a COP approach for the motion generation, but the skills encoded in the FSM states are defined as a configuration of a traditional trajectory generator. However, the findings presented in Chapter 3 and Chapter 4 can be applied to the Preview Coordination formalism.

Compared to the *SDG*, the Preview Coordination is affected by some limitations due to the *procedural* nature of a FSM model with respect to the *declarative* solution provided by the *uSDL*. Concretely, the situations that the Preview Coordination can represent are limited by the expressivity of a FSM: the activities are scheduled sequentially or concurrently, without explicit logical constraints defined along the execution of an activity. Another limitation is the absence of an explicit primitive to model non-nominal situations: the application developer that encode the FSM model must consider all the possible combination of failures, which is non-trivial for complex applications. As a consequence, the skill composability is limited to a subset of cases with respect to the *SDG*.

7.3 Suggestions for Future Work

Conducting research, proposing solutions to the addressed research questions lead to novel unsolved matters. This dissertation is not an exception, and this section provides few ideas on the research direction that could be carried out in further investigations.

Skill prototypes database
A natural continuation of the work carried out in this dissertation is the development of an extensive database of skill prototypes, in the same vein as the examples provided in Chapter 4. In detail, the skills implemented are kinematic-based only, but force-based skills would improve (and sometimes simplify) actions that require physical interaction.

Learning-based approaches for Skill Prototypes
This research focuses on *programmable skills*, and the Chapter 4 shows many examples of *programmable* skill prototypes. This approach requires modelling efforts and development time of a "skill developer expert", even if a set of few skill prototypes already covers many practical scenarios. A complementary research direction regards *learnable skills*, and further investigations can be addressed to integrate both methods. Concrete suggestions depends on the type of learning over existing skill prototypes:

- given a skill prototype, to learn and to improve the execution of skill instances from previous executions in the same context. This learning consists in determine some parameters of the COP previously specified manually, such as control gains and constraint weights;
- matching human demonstrations with the composition of existing skill prototypes, thus learning the relationships between multiple skills that perform a certain effect;
- among multiple skill prototypes that suit to the same context, selecting the most appropriate one by recurrent execution and evaluation; in this case, the simplest mechanism of learning is *caching* the success of a previous execution;
- discovering new patterns as a composition of existing skill prototypes from recurrent executions (and the context in which they are employable).

Model Verification and Validation
Chapter 2 and Chapter 6 provide the structural constraints that determine the well-formedness of the *SDG* model (*e.g.*, acyclicity of the graph). Furthermore, the model is deterministic: the execution outcome does not change if the initial conditions are the same, as well as the truth value of the logical conditions between two executions[6]. To aid the skill prototype developer, a suggested future work is to provide formal tools and software support for verification and validation of a *SDG* model.

Software Tools for Developing Skills
This dissertation focuses on the formalisation and the novelty of the proposed framework. It is worth to mention that the author developed a few tools to aid the online visualisation of the deployed *SDG* and the running skills in a web-based Graphical User Interface (GUI). However, these tools are not mature and not sufficient to fully exploit the potential of the skill-based

[6]Note that the *SDG* model is deterministic, but not the context of the execution. Here it is assumed that the context of the execution is invariant. Within these assumption, model verification and validation aids the skill development.

approach, *e.g.*, it is not possible to compose graphically the skills, but text-based only. The development of tools is a necessary step to disseminate the presented methodology, and some functionalities must not be implemented at the *SDG* level, but at its structural level (*i.e.*, *NPC4*), such that those efforts also hold for other graph-based methodologies.

Distributed *SDG*
The software infrastructure that performs the experiments in this dissertation is centralised: the task specification is translated to a single COP, and a numerical solver computes the control action for one robotic platform only. In the same vein, the *SDG* model is deployed monolithically in one single *SDG*-E. However, the proposed models are ready to be deployed in a distributed architecture. For instance, a dual arm manipulator (already employed in the experiments) can be seen as a couple of agents that perform several actions, whether they are competitive or not. In this case, the whole problem is formulated as single COP, but the task specification can be deployed in two separate COP(s), while the execution monitoring and the coordination driven by the *SDG*-E is still in common. Likewise, the *SDG* model can be deployed in two different *SDG*-E, each running on a different robotic platform. The primitives in the *SDG* model allow this feature, since it is possible to represent a *non-controlled* skill: in a “local” *SDG* model, someone else’s duty is represented in this way, and the dependencies “bridge” between the separate models. However, the implementation of a distributed *SDG* raises a set of non-trivial new challenges, such as the capability of maintaining the distributed models coherent and up-to-date.

Integrating with Symbolic Planners
This work illustrates a formal methodology to ground symbolic plans and actions into to an executable *SDG* model. Chapter 2 presents all the elements to link to existing planners, while the overall dissertation provides a systematic way to describe situations in both continuous and discrete domains. However, the integration with a symbolic planner is not addressed in this work, and further investigations can be carried out in this direction. As an additional hint, the most promising approaches are the integration with classical planners, and the integration with Hierarchical Task Networks (NTH): the former would allow to solve a large variety of planning problems; the latter is widely used for fast reactive planning and backtracking. Another concrete suggestion is to consider the geometric constraints used in the symbolic planner [89, 88] to generate *SDG* models instead of computing a path to follow.

Improvements on Solvers and Execution Monitor
The Robotics Community is already focused on this aspect, but improvements on the continuous problem definition and the numerical solvers are still a hot research topic, *e.g.*, solvers that combines both kinematic and dynamic

constraints. Moreover, the execution monitoring and the evaluation of the Quality of Service (QoS) of a skill execution requires some further investigations; some preliminary work is already carried out by the author of this dissertation [136].

Geometric-based Task DSL as a Benchmark Language for Constraint-based Whole Body Control Solvers
The Planning Domain Definition Language (PDDL) [56, 59] serves the purpose of benchmark language between symbolic planners. In fact, it is developed to formulate planning problems for the International Planning Competition (IPC), also indicating which features the solver must implement (*i.e.*, the solver capabilities) to solve a specific problem. In the same vein, the proposed geometric-based task DSL can cover the same role of benchmark language for constraint-based approaches for Robotics. The DSL in Chapter 3 can be employed in multiple framework already (*e.g.*, eTaSL [2], SoT [96]), and the choice over the JSON format simplifies the parsing of the specification. This standardisation effort, together with a broad adoption among the Robotics Community, could help to measure the advancements that, so far, are validated without a scientific comparison with respect to alternatives applied to the same context.

Appendix A

JSON and JSON-Schema Models

A.1 JSON Schema *uSDL* Meta Model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/sdg-v01",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type" : "object",
  "properties" : {
    "metamodel" : { "enum" : ["http://people.mech.kuleuven.be↩
        /~u0072295/sdg-v01"] },
    "skills" : {
      "type" : "array",
      "minItems" : 1,
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/skill" }
    },
    "contains" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/container" }
    },
    "conditions" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/condition" }
    },
    "dependencies" : {
      "type" : "array",
      "uniqueItems" : true,
      "items" : { "$ref" : "#/definitions/dependency" }
    }
  },
  "required" : [ "metamodel", "skills", "conditions", "↩
      dependencies"],
  "additionalProperties" : false,
  "definitions" :{
    "skill" : {
      "type" : "object",
      "properties" : {
        "type" : { "enum" : ["skill"] },
        "id"   : { "type" : "string" },
        "eff"  : { "type" : "string" },
        "side-eff" : {
          "type" : "array",
          "uniqueItems" : true,
          "minItems" : 1,
          "items" : { "type" : "string" }
        }
      },
      "additionalProperties" : false,
      "required" : [ "type", "id" ]
    },
    "container" : {
      "type" : "object",
      "properties" : {
        "parent" : { "type" : "string" },
        "children" : { "type" : "array", "items" : { "type" : ↩
            "string"}, "minItems" : 1, "uniqueItems" : true }
      }
    },
    "condition" : {
      "type" : "object",
      "properties" : {
        "type" : { "enum" : ["condition"] },
        "id"   : { "type" : "string" },
        "monitor" : { "oneOf" : [
          { "type" : "object",
            "properties" : {
              "type" : { "enum" : ["function"]},
              "id" : { "type" : "string" }
            },
            "required" : [ "type", "id"],
            "additionalProperties" : false
          },
          { "type" : "object",
            "properties" : {
              "type" : { "enum" : ["expr"]},
              "expr" : { "$ref" : "#/definitions/lexpr" }
            },
            "required" : [ "type", "expr"],
            "additionalProperties" : false
          }
          ]}
        },
        "additionalProperties" : false,
        "required" : [ "type", "id", "monitor" ]
      },
      "dependency" : {
        "type" : "object",
        "properties" : {
          "type" : { "enum" : ["dependency"] },
          "id"   : { "type" : "string" },
          "relationship" : { "enum" : ["toStart", "continuesIf↩
              ", "latches"] },
          "condition"  : { "type" : "string" },
          "guard" : {
            "oneOf" : [
              { "$ref": "#/definitions/lexpr" },
              { "type" : "string" }
              ]},
              "requiredby" : {
                "type" : "array",
                "uniqueItems" : true,
                "minItems" : 1,
                "items" : { "type" : "string" }
              }
            },
            "additionalProperties" : false,
            "required" : [ "type", "id", "relationship", "↩
                condition" ]
          },
          "lexpr" : {
            "type" : "object",
            "properties" : {
              "operator": { "enum" : ["and", "or", "neg", "nor↩
                  ", "nand" ] },
              "args": {
                "type" : "array",
                "minItems": 2,
                "maxItems": 2,
                "items" : { "oneOf" : [
                  { "type" : "string"},
                  { "$ref" : "#/definitions/lexpr" }
                  ]}
                }
              }
            }
          }
        }
```

Listing A.1: JSON-Schema meta model of the *uSDL* model

A.2 Geometric-based Task DSL

A.2.1 Geometric Primitives

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      point#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Point Entity",
  "type": "object",
  "properties": {
    "x": {
      "type": "number",
      "description": "coordinate along x-axis"
    },
    "y": {
      "type": "number",
      "description": "coordinate along y-axis"
    },
    "z": {
      "type": "number",
      "description": "coordinate along z-axis"
    },
    "description" : { "type" : "string" },
    "type": { "enum": ["point"] }
  },
  "required": [ "x", "y", "z", "type" ],
  "additionalProperties": false
}
```

Listing A.2: Point JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      line#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Line Entity",
  "type": "object",
  "properties": {
     "type": { "enum": ["line"] },
     "description" : { "type" : "string" },
     "direction": { "$ref": "http://people.mech.kuleuven.be/~↩
         u0072295/jgeom_constr/versor#" },
     "p": { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
         jgeom_constr/point#" }
   },
   "required": [ "direction", "p", "type" ],
   "additionalProperties": false
}
```

Listing A.3: Line JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      versor#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Versor Entity",
  "type": "object",
  "properties": {
    "type": { "enum": [ "versor"] },
    "description" : { "type" : "string" },
    "x": {
      "type": "number",
      "description": "coordinate along x-axis"
     },
    "y": {
      "type": "number",
      "description": "coordinate along y-axis"
    },
    "z": {
      "type": "number",
      "description": "coordinate along z-axis"
    }
  },
  "required": [ "x", "y", "z", "type" ],
  "additionalProperties": false
}
```

Listing A.4: Versor JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      plane#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Plane Entity",
  "type": "object",
  "properties": {
     "type": { "enum": ["plane"] },
     "description" : { "type" : "string" },
     "normal": { "$ref": "http://people.mech.kuleuven.be/~↩
         u0072295/jgeom_constr/versor#" },
     "p": { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
         jgeom_constr/point#" }
  },
  "required": [ "normal", "p", "type" ],
  "additionalProperties": false
}
```

Listing A.5: Plane JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      primitive#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Geometric Primitive",
  "type": "object",
  "properties": {
    "description" : { "type" : "string" },
    "reference_frame": {
      "type": "string",
      "description": "reference frame for the geometric entity"
    },
    "entity" : { "$ref": "#/definitions/entity" }
  },
  "required": [ "reference_frame", "entity" ],
  "additionalProperties": false,
  "definitions": {
    "entity": {
        "oneOf": [
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/point#" },
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/versor#" },
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/plane#" },
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/line#" }
        ]
    }
  }
}
```

Listing A.6: Primitive JSON-Schema meta model

A.2.2 Constraints based on Expressions

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Geometric expression between two geometric ↩
      primitives",
  "type": "object",
  "properties": {
    "description":{"type": "string" },
    "type":{"enum":["geometric_expression"] },
    "expression_type":{"enum":[
      "point_point_distance",
      "line_point_distance",
      "projection_point_on_line",
      "line_line_distance",
      "plane_point_distance",
      "angle_btw_planes",
      "angle_btw_versors",
      "angle_plane_versor",
      "angle_line_point"
      ]
    },
    "expression_name": {"type": "string" },
    "primitive_1":{"$ref": "http://people.mech.kuleuven.be/~↩
        u0072295/jgeom_constr/primitive#" },
    "primitive_2":{"$ref": "http://people.mech.kuleuven.be/~↩
        u0072295/jgeom_constr/primitive#" }
  },
  "required": ["expression_type", "expression_name", "↩
      primitive_1", "primitive_2", "type" ],
  "additionalProperties":false,
  "oneOf":[
   {"properties":{
    "expression_type":{"enum":["point_point_distance" ] },
    "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
        properties":{"type":{"enum":["point"]}}}]}}},
    "primitive_2":{"properties":{"entity":{"oneOf":[{"↩
        properties":{"type":{"enum":["point"]}}}]}}}
    }
   },
   {"properties" :{
      "expression_type":{"enum":["angle_btw_versors" ] },
       "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["versor"]}}}]}}},
       "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["versor"]}}}]}}}
     }
   },
   {"properties":{
        "expression_type":{"enum":["line_line_distance" ] },
        "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["line"]}}}]}}},
        "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["line"]}}}]}}}
      }
   },
   {"properties":{
        "expression_type":{"enum":["angle_btw_planes" ] },
        "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["plane"]}}}]}}},
        "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["plane"]}}}]}}}
      }
   },
   {"oneOf":[
     {
       "properties":{
         "expression_type":{"enum":["line_point_distance", "↩
             projection_point_on_line", "angle_line_point" ] },
         "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
             properties":{"type":{"enum":["line"]}}}]}}},
         "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
             properties":{"type":{"enum":["point"]}}}]}}}
       }
    },
    {
      "properties":{
        "expression_type":{"enum":["line_point_distance", "↩
            projection_point_on_line", "angle_line_point" ] },
        "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["point"]}}}]}}},
        "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["line"]}}}]}}}
      }
    }
  ]},
  {"oneOf":[
    {
      "properties":{
        "expression_type":{"enum":["plane_point_distance"] },
        "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["plane"]}}}]}}},
        "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
            properties":{"type":{"enum":["point"]}}}]}}}
      }
   },
   {
     "properties":{
       "expression_type":{"enum":["plane_point_distance"] },
       "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["point"]}}}]}}},
       "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["plane"]}}}]}}}
     }
   }
 ]},
 {"oneOf":[
   {
     "properties":{
       "expression_type":{"enum":["angle_plane_versor"] },
       "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["plane"]}}}]}}},
       "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
           properties":{"type":{"enum":["versor"]}}}]}}}
     }
  },
  {
    "properties":{
      "expression_type":{"enum":["angle_plane_versor"] },
      "primitive_1":{"properties":{"entity":{"oneOf":[{ "↩
          properties":{"type":{"enum":["versor"]}}}]}}},
      "primitive_2":{"properties":{"entity":{"oneOf":[{ "↩
          properties":{"type":{"enum":["plane"]}}}]}}}
    }
  } ] } ] }
```

Listing A.7: Geometric expression JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      single_joint_expression#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "expression that returns the value of a single↩
       joint",
  "type": "object",
  "properties": {
    "description" : { "type": "string" },
    "type" : { "enum" : [ "single_joint_expression"] },
    "expression_name": { "type": "string" },
    "joint_name": { "type": "string" }
  },
  "required": [ "expression_name", "joint_name", "type" ],
  "additionalProperties" : false
}
```

Listing A.8: Joint expression JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      expression#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Expression",
  "type": "object",
  "properties": {
    "description" : { "type" : "string" },
    "expression" : { "$ref": "#/definitions/expression" }
  },
  "required": [ "expression" ],
  "additionalProperties": false,
  "definitions": {
    "expression": {
        "oneOf": [
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/geometric_expression#" },
          { "$ref": "http://people.mech.kuleuven.be/~u0072295/↩
              jgeom_constr/single_joint_expression#" }
        ]
    }
  }
}
```

Listing A.9: Expression JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      constraint#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Constraint over an expression",
  "type": "object",
  "properties": {
    "description" : { "type": "string" },
    "type" : { "enum" : ["constraint"] },
    "constraint_name": { "type": "string" },
    "expression" : { "$ref" : "http://people.mech.kuleuven.be/~↩
        u0072295/jgeom_constr/expression#" },
    "behaviour" : { "$ref": "http://people.mech.kuleuven.be/~↩
        u0072295/jgeom_constr/behaviour#" }
  },
  "required": [ "constraint_name", "expression", "behaviour", "↩
      type" ],
  "additionalProperties" : false
}
```

Listing A.10: Constraint JSON-Schema meta model

A.2.3 Behaviour, Task and Monitors

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      monitor",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Monitor over an expression",
  "type": "object",
  "oneOf": [
    { "$ref" : "#/definitions/single_bound"},
    { "$ref" : "#/definitions/double_bound"}
  ],
  "definitions" : {
    "common" : {
      "properties" : {
        "type" : { "enum" : ["monitor"] },
        "description" : { "type": "string" },
        "monitor_name": { "type": "string" },
        "event_risen":  { "type": "string" },
        "monitor_var_type" : { "enum": ["pos", "for", "vel"] },
        "monitored_expression" : { "$ref": "http://people.mech.↩
            kuleuven.be/~u0072295/jgeom_constr/expression#" }
      },
      "required" : [ "type", "monitor_name", "event_risen", "↩
          monitor_var_type", "monitored_expression" ]
    },
    "single_bound" : {
      "allOf" : [
        { "$ref" : "#/definitions/common" },
        { "properties" : {
            "comparison_type"  : { "enum" : ["less", "more"] },
            "bound" : { "type": "number" }
          },
          "required": [ "bound", "comparison_type" ]
        }
      ]
    },
    "double_bound" : {
      "allOf" : [
        { "$ref" : "#/definitions/common" },
        { "properties" : {
            "comparison_type"  : { "enum" : ["in_interval", "↩
                out_interval"] },
            "lower_bound" : { "type": "number" },
            "upper_bound" : { "type": "number" }
          },
          "required": [ "lower_bound", "upper_bound",  "↩
              comparison_type" ]
        }
      ]
    }  }  }
```

Listing A.11: Monitor JSON-Schema meta model

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      behaviour#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Control Behaviour for a target constraint",
  "type": "object",
  "properties": {
    "description" : { "type": "string" },
    "type" : { "enum": ["behaviour"] },
    "behaviour_type" : { "enum" : [
      "positioning",
      "move_toward",
      "interaction",
      "compliant",
      "position_limit",
      "velocity_limit",
      "force_limit"
    ]
    }
  },
  "required": [ "behaviour_type", "type" ],
  "oneOf" : [
    { "properties" : {
        "behaviour_type" : { "enum" : [ "position_limit", "↩
            velocity_limit", "force_limit" ] },
        "traj_gen_upper" : { "$ref" : "http://people.mech.↩
            kuleuven.be/~u0072295/jgeom_constr/trajectory" },
        "traj_gen_lower" : { "$ref" : "http://people.mech.↩
            kuleuven.be/~u0072295/jgeom_constr/trajectory" }
      },
      "required" :  [ "traj_gen_upper", "traj_gen_lower" ]
    },
    { "properties" : {
        "behaviour_type" : { "enum" : [ "positioning", "↩
            move_toward", "interaction", "compliant" ] },
        "specification" : { "type" : "number" },
        "traj_gen" : { "$ref" : "http://people.mech.kuleuven.be↩
            /~u0072295/jgeom_constr/trajectory" }
      },
      "required" :  [ "traj_gen", "specification" ]
    }
  ]
}
```

Listing A.12: Behaviour JSON-Schema meta model, linked to a Constraint

```
{
  "id": "http://people.mech.kuleuven.be/~u0072295/jgeom_constr/↩
      task#",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Constraint-based Task using Geometric ↩
      Relations",
  "type": "object",
  "properties": {
    "description" : { "type": "string" },
    "type" : { "enum" : ["geom_task"] },
    "task_name": { "type": "string" },
    "primary" : {
      "type" : "array",
      "items" : { "$ref" : "http://people.mech.kuleuven.be/~↩
          u0072295/jgeom_constr/constraint" },
      "uniqueItems" : true
    },
    "auxiliary" : {
      "type" : "array",
      "items" : { "$ref" : "http://people.mech.kuleuven.be/~↩
          u0072295/jgeom_constr/constraint" },
      "uniqueItems" : true
    },
    "safety" : {
      "type" : "array",
      "items" : { "$ref" : "http://people.mech.kuleuven.be/~↩
          u0072295/jgeom_constr/constraint" },
      "uniqueItems" : true
    },
    "monitors" : {
       "type": "array",
       "items" : { "$ref" : "http://people.mech.kuleuven.be/~↩
           u0072295/jgeom_constr/monitor" },
       "minItems" : 1,
       "uniqueItems" : true
    }
  },
  "required": [ "task_name", "primary", "auxiliary", "safety", ↩
      "monitors", "type" ],
  "additionalProperties" : false
}
```

Listing A.13: Task JSON-Schema meta model (composition of constraints and monitors)

A.3 Spreading Task Specification

```
{
  "type":"geom_task",
  "monitors":[ {
      "monitored_expression":{
        "expression":{
          "type":"geometric_expression",
          "description":"distance spoon_tip (XY) on the dough ↩
              radius from the center",
          "primitive_1":{
            "entity":{ "type":"point",
              "y":0.01, "x":0.23, "z":0
            },
            "reference_frame":"rightarm_gripper"
          },
          "primitive_2":{
            "entity":{
              "type":"line",
              "direction":{ "type":"versor",
                "y":0, "x":1, "z":0
              },
              "p":{ "type":"point",
                "y":0, "x":0, "z":0
              }
            },
            "reference_frame":"center"
          },
          "expression_type":"projection_point_on_line",
          "expression_name":"radius_distance"
        }
      },
      "type":"monitor",
      "comparison_type":"in_interval",
      "monitor_name":"is_spoon_on_perifery",
      "lower_bound":0.08,
      "monitor_var_type":"pos",
      "event_risen":"e_spoon_on_perifery",
      "upper_bound":0.12
    }
  ],
  "task_name":"spread_action",
  "primary":[ {
      "type":"constraint",
      "description":"tip spoon (XY) constrainted to lie on ↩
          dought area",
      "constraint_name":"spoon_along_dough_radius",
      "behaviour":{
        "traj_gen":{
          "type":"trajectory",
          "traj_type":"trapezoidal_with_duration",
          "target":0.1, "duration":5,
          "vel_max":0.1, "acc_max":0.1
        },
        "type":"behaviour",
        "specification":1.5,
        "behaviour_type":"positioning"
      },
      "expression":{
        "expression":{
          "type":"geometric_expression",
          "description":"distance spoon_tip (XY) on the dough ↩
              radius from the center",
          "primitive_1":{
            "entity":{ "type":"point",
              "y":0.01,  "x":0.23, "z":0
            },
            "reference_frame":"rightarm_gripper"
          },
          "primitive_2":{
            "entity":{
              "type":"line",
              "direction":{ "type":"versor",
                "y":0, "x":1, "z":0
              },
              "p":{ "type":"point",
                "y":0, "x":0, "z":0
              }
            },
            "reference_frame":"center"
          },
          "expression_type":"projection_point_on_line",
          "expression_name":"radius_distance"
        }
      }
    },
    {
      "type":"constraint",
      "description":"force constraint on Z-direction wrt table↩
          , imposed contact spoon-table",
      "constraint_name":"force_contact_spoon_on_table",
      "behaviour":{
        "traj_gen":{
          "target":-2,
          "type":"trajectory",
          "traj_type":"constant"
        },
        "type":"behaviour",
        "specification":0.02,
        "behaviour_type":"interaction"
      },
      "expression":{
        "expression":{
          "type":"geometric_expression",
          "description":"force imposed on the table, Z-↩
              direction",
          "primitive_1":{ "type":"point",
            "entity":{
              "y":0.01, "x":0.23, "z":0
            },
            "reference_frame":"rightarm_gripper"
          },
          "primitive_2":{
            "entity":{
              "type":"line",
              "direction":{ "type":"versor",
                "y":0, "x":0, "z":1
              },
              "p":{ "type":"point",
                "y":0, "x":0, "z":0
              }
            },
            "reference_frame":"center"
          },
          "expression_type":"projection_point_on_line",
          "expression_name":"force_z_on_table"
        }
} } ] }
```

Listing A.14: Task specification of *"spread with spoon"*. For the sake of brevity, safety and auxiliary constraints are omitted.

Bibliography

[1] Abdellatif, T., Bensalem, S., Combaz, J., de Silva, L., and Ingrand, F. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems 60*, 12 (2012), 1563–1578.

[2] Aertbeliën, E., and De Schutter, J. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Chicago, IL, USA, 2014), IROS2014, pp. 1540–1546.

[3] Alami, R., Chatila, R., Fleury, R., Ghallab, M., and Ingrand, F. An architecture for autonomy. *The International Journal of Robotics Research 17*, 4 (1998), 315–337.

[4] Allen, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM 26*, 11 (1983), 832–843.

[5] Allen, J. F., and Ferguson, G. Actions and events in interval temporal logic. *Journal of logic and computation 4*, 5 (1994), 531–579.

[6] Allen, R. R., and Dubowsky, S. Mechanisms as components of dynamic systems: A Bond Graph approach. *Journal of Electronic Imaging* (1977), 104–111.

[7] Argall, B. D., Chernova, S., Veloso, M., and Browning, B. A survey of robot learning from demonstration. *Robotics and Autonomous Systems 57* (2009), 469–483.

[8] Atkinson, C., and Kühne, T. Model-driven development: a metamodeling foundation. *IEEE software 20*, 5 (2003), 36–41.

[9] Avanzini, G. B., Zanchettin, A. M., and Rocco, P. Reactive constrained model predictive control for redundant mobile manipulators. In *Intelligent Autonomous Systems 13 - Proceedings of the 13th International Conference IAS-13, Padova, Italy, July 15-18, 2014* (2014), pp. 1301–1314.

[10] Baerlocher, P., and Boulic, R. Task-priority formulations for the kinematic control of highly redundant articulated structures. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vancouver, British Columbia, Canada, 1998), IROS98, pp. 323–329.

[11] Baier, C., Katoen, J.-P., et al. *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[12] Bartels, G., Kresse, I., and Beetz, M. Constraint-based movement representation grounded in geometric features. In *13th IEEE-RAS International Conference on Humanoid Robots* (Atlanta, Georgia, USA, October 15–17 2013).

[13] Basu, A., Bozga, M., and Sifakis, J. Modeling heterogeneous real-time components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods* (2006), SEFM '06, IEEE Computer Society, pp. 3–12.

[14] Beetz, M., Mösenlechner, L., and Tenorth, M. CRAM—A cognitive robot abstract machine for everyday manipulation in human environments. In *Proceedings of the 2010 International Conference on Advanced Robotics* (2010), pp. 1012–1017.

[15] Belleghem, K. V., Denecker, M., and Schreye, D. D. On the relation between situation calculus and event calculus. *The Journal of Logic Programming 31*, 1–3 (1997), 3–37.

[16] Bennett, B., and Galton, A. P. A unifying semantics for time and events. *Artificial Intelligence 153*, 1–2 (2004), 13–48.

[17] Bensalem, S., de Silva, L., Ingrand, F., and Yan, R. A verifiable and correct-by-construction controller for robot functional levels. *Journal of Software Engineering in Robotics 2*, 1 (2011), 1–19.

[18] Berenson, D., Srinivasa, S., and Kuffner, J. Task space regions: A framework for pose-constrained manipulation planning. *The International Journal of Robotics Research 30*, 12 (October 2011), 1435 – 1460.

[19] Bézivin, J. On the unification power of models. *Software and Systems Modeling 4*, 2 (2005), 171–188.

[20] Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Bruyninckx, H., Soetens, P., Hägele, M., Pott, A., Breedveld, P., Broenink, J., Brugali, D., and Tomatis, N. BRICS—Best practice in robotics. In *41st International Symposium on Robotics* (Munich, Germany, 2010), pp. 968–975.

[21] Blumenthal, S., and Bruyninckx, H. Towards a domain specific language for a scene graph based robotic world model. In *4th International Workshop on Domain-Specific Languages and models for ROBotic systems* (2013).

[22] Blumenthal, S., Bruyninckx, H., Nowak, W., and Prassler, E. A scene graph based shared 3D world model for robotic applications. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 453–460.

[23] Bøgh, S., Nielsen, O. S., Pedersen, M. R., Krüger, V., and Madsen, O. Does your robot have skills? In *The 43rd Intl. Symp. on Robotics (ISR2012)* (2012).

[24] Bonasso, R. P., Kortenkamp, D., Miller, D. P., and Slack, M. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence 9* (1995), 237–256.

[25] Borghesan, G., Aertbeliën, E., and De Schutter, J. Constraint- and synergy-based specification of manipulation tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Hong Kong, 2014).

[26] Borghesan, G., Scioni, E., Kheddar, A., and Bruyninckx, H. Introducing geometric constraint expressions into robot constrained motion specification and control. *IEEE Robotics and Automation Letters PP*, 99 (2015), 1–1.

[27] Borghesan, G., Willaert, B., and De Schutter, J. A constraint-based programming approach to physical human-robot interaction. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Minnesota, USA, 2012), pp. 3890–3896.

[28] Bray, T. RFC 7159, The JavaScript Object Notation (JSON) Data Interchange Format, August 2014.

[29] Bruyninckx, H. Open robot control software: the OROCOS project. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation* (Seoul, Korea, 2001), ICRA2001, pp. 2523–2528.

[30] Bruyninckx, H., and De Schutter, J. Specification of force-controlled actions in the "Task Frame Formalism": A synthesis. *IEEE Transactions on Robotics and Automation 12*, 5 (1996), 581–589.

[31] Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. The BRICS Component Model: A model-based development paradigm for complex robotics software systems. In *28th ACM Symposium On Applied Computing* (2013), pp. 1758–1764.

[32] Chein, M., and Mugnier, M.-L. *Graph-based knowledge representation: computational foundations of conceptual graphs.* Springer Science & Business Media, 2008.

[33] Colledanchise, M., and Ogren, P. How behavior trees modularize robustness and safety in hybrid systems. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Chicago, IL, USA, 2014), IROS2014, pp. 1482–1488.

[34] Cutkosky, M., and Wright, P. Modeling manufacturing grips and correlations with the design of robotic hands. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on* (Apr 1986), vol. 3, pp. 1533–1539.

[35] De Schutter, J. Invariant description of rigid body motion trajectories. *Transactions of the ASME, Journal of Mechanisms and Robotics 2*, 1 (2010), 011004/1–9.

[36] De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., and Bruyninckx, H. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research 26*, 5 (2007), 433–455.

[37] de Silva, L., Pandey, A., and Alami, R. An interface for interleaved symbolic-geometric planning and backtracking. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Tokyo, Japan, Nov 2013), IROS2013, pp. 232–239.

[38] Debrouwere, F., Van Loock, W., Pipeleers, G., Tran Dinh, Q., Diehl, M., De Schutter, J., and Swevers, J. Time-optimal path following for robots with convex-concave constraints using sequential convex programming. *IEEE Transactions on Robotics 29*, 6 (2013), 1485–1495.

[39] Dechter, R. Tractable structures for constraint satisfaction problems. *Handbook of constraint programming*, part I (2006), 209–244.

[40] Decré, W., , Bruyninckx, H., and De Schutter, J. An optimization-based estimation and adaptive control approach for human-robot cooperation. In *12th International Symposium on Experimental Robotics* (Delhi, India, 2010).

[41] Decré, W., Smits, R., Bruyninckx, H., and De Schutter, J. Extending iTaSC to support inequality constraints and non-instantaneous task specification. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (Kobe, Japan, 2009), ICRA2009, pp. 964–971.

[42] Del Prete, A., Nori, F., Metta, G., and Natale, L. Prioritized motion–force control of constrained fully-actuated robots: "task space inverse dynamics". *Robotics and Autonomous Systems 63* (2015), 150–157.

[43] Del Prete, A., Romano, F., Natale, L., Metta, G., Sandini, G., and Nori, F. Prioritized optimal control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Hong Kong, 2014), pp. 2540–2545.

[44] Doty, K. L., Melchiorri, C., and Bonivento, C. A theory of generalized inverses applied to robotics. *The International Journal of Robotics Research 12*, 1 (1993), 1–19.

[45] Dromey, R. From requirements to design: formalizing the key steps. In *First International Conference on Software Engineering and Formal Methods.* (sept. 2003), pp. 2 –11.

[46] Elfring, J., van den Dries, S., van de Molengraft, M. J. G., and Steinbuch, M. Semantic world modeling using probabilistic multiple hypothesis anchoring. *Robotics and Autonomous Systems 61*, 2 (2013), 95–105.

[47] Erdem, E., Haspalamutgil, K., Palaz, C., Patoglu, V., and Uras, T. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Shangai, China, 2011), ICRA2011, pp. 4575–4581.

[48] Escande, A., Mansard, N., and Wieber, P.-B. Hierarchical quadratic programming: Fast online humanoid-robot motion generation. *The International Journal of Robotics Research* (June 2014), pp. 1006–1028.

[49] Ferreau, H., Kirches, C., Potschka, A., Bock, H., and Diehl, M. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation 6*, 4 (2014), 327–363.

[50] Fikes, R. E. Monitored execution of robot plans produced by STRIPS. In *IFIP Congress* (1971).

[51] Fikes, R. E., and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 3/4 (1971), 189–208.

[52] Finkemeyer, B., Kröger, T., and Wahl, F. M. Executing assembly tasks specified by manipulation primitive nets. *Advanced Robotics 19*, 5 (2005), 591–611.

[53] Firby, R. J. Task networks for controlling continuous processes. In *AIPS* (1994), pp. 49–54.

[54] Firby, R. J., and Slack, M. G. Task execution: Interfacing to reactive skill networks. In *AAAI Spring Symposium* (1995).

[55] Fowler, M. *Domain Specific Languages.* Addison-Wesley Professional, 2010.

[56] Fox, M., and Long, D. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20* (2003), 61–124.

[57] Galiegue, F., and Zyp, K. JSON Schema, Draft 0.4v, August 2013.

[58] Gastin, P., and Oddoux, D. Fast ltl to büchi automata translation. In *Computer Aided Verification* (2001), Springer, pp. 53–65.

[59] Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence 173*, 5-6 (2009), 619–668.

[60] GeRT. Generalizing robot manipulation tasks: The GeRT Project. `http://www.gert-project.eu/`.

[61] Ghallab, M., Nau, D., and Traverso, P. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence 208* (2014), 1–17.

[62] Gheţa, I., Heizmann, M., Belkin, A., and Beyerer, J. World modeling for autonomous systems. In *KI 2010: Advances in Artificial Intelligence.* Springer, 2010, pp. 176–183.

[63] Glass, B., Cannon, H., Branson, M., Hanagud, S., and Paulsen, G. Dame: planetary-prototype drilling automation. *Astrobiology 8*, 3 (2008), 653–664.

[64] Grisetti, G., Kummerle, R., Stachniss, C., and Burgard, W. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine 2*, 4 (2010), 31–43.

[65] Guo, M., Johansson, K. H., and Dimarogonas, D. V. Motion and action planning under ltl specifications using navigation functions and action description language. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Tokyo, Japan, 2013), IROS2013, pp. 240–245.

[66] Harel, D. State charts: A visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[67] Hasegawa, T., Suehiro, T., and Takase, K. A model-based manipulation system with skill-based execution. *IEEE Transactions on Robotics and Automation 8*, 1 (1992), 535–544.

[68] Holz, D., Topalidou-Kyniazopoulou, A., Rovida, F., Pedersen, M., Kruger, V., and Behnke, S. A skill-based system for object perception and manipulation for automating kitting tasks. In *IEEE International Conference on Emerging Technologies and Factor Automation* (Luxemburg, September 2015), pp. 1–9.

[69] Hähnel, D., Burgard, W., and Lakemeyer, G. Golex—bridging the gap between logic (golog) and a real robot. In *KI-98: Advances in Artificial Intelligence*, O. Herzog and A. Günter, Eds., vol. 1504 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 165–176.

[70] Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. Lua—an extensible extension language. *Softw. Pract. Exper. 26*, 6 (1996), 635–652.

[71] Ingrand, F., and Ghallab, M. Robotics and artificial intelligence: A perspective on deliberation functions. *AI Communications 27*, 1 (2014), 63–80.

[72] Ingrand, F. F., Chatila, R., Alami, R., and Robert, F. PRS: a high level supervision and control language for autonomous mobile robots. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation* (Minneapolis, MN, 1996), ICRA96, pp. 43–49.

[73] JAIN, A., AND KEMP CHARLES, C. Pulling open doors and drawers: Coordinating an omni-directional base and a compliant arm with equilibrium point control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Anchorage, Alaska, USA, 2010), pp. 1807–1814.

[74] KAELBLING, L. P., AND LOZANO-PÉREZ, T. Integrated task and motion planning in belief space. *The International Journal of Robotics Research 32*, 9-10 (2013).

[75] KALLMANN, M., AND JIANG, X. A motion planning framework for skill coordination and learning. In *Motion Planning for Humanoid Robots*. Springer, 2010, pp. 277–306.

[76] KARAYIANNIDIS, Y., SMITH, C., BARRIENTOS, F., OGREN, P., AND KRAGIC, D. An adaptive control approach for opening doors and drawers under uncertainties. *Robotics, IEEE Transactions on 32*, 1 (Feb 2016), 161–175.

[77] KARAYIANNIDIS, Y., SMITH, C., VINA, F., OGREN, P., AND KRAGIC, D. Mdel-free robot manipulation of doors and drawers by means of fixed-grasps. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on* (May 2013), pp. 4485–4492.

[78] KEITH, F., MANSARD, N., MIOSSEC, S., AND KHEDDAR, A. Optimization of tasks warping and scheduling for smooth sequencing of robotic actions. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (St. Louis, Missouri, 2009), IROS2009, pp. 1609–1614.

[79] KEITH, F., WIEBER, P.-B., MANSARD, N., AND KHEDDAR, A. Analysis of the discontinuities in prioritized tasks-space control under discreet task scheduling operations. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (San Francisco, California, 2011), IROS2011, pp. 3887–3892.

[80] KENT, S. Model driven engineering. In *Integrated Formal Methods*, vol. 2335 of *Lecture Notes in Computer Science*. 2002, pp. 286–298.

[81] KLOTZBÜCHER, M., AND BRUYNINCKX, H. Coordinating robotic tasks and systems with rFSM Statecharts. *Journal of Software Engineering in Robotics 3*, 1 (2012), 28–56.

[82] KLOTZBÜCHER, M., SMITS, R., BRUYNINCKX, H., AND DE SCHUTTER, J. Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages. In *Proceedings of the 2011*

IEEE/RSJ International Conference on Intelligent Robots and Systems (San Francisco, California, 2011), IROS2011, pp. 4684–4689.

[83] Kortenkamp, D., and Simmons, R. G. Robotic systems architectures and programming. In *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 187–206.

[84] Kowalski, R., and Sadri, F. Reconciling the event calculus with the situation calculus. *The Journal of Logic Programming 31*, 1–3 (1997), 39–58.

[85] Kowalski, R., and Sergot, M. A logic-based calculus of events. *New Gen. Comput. 4*, 1 (January 1986), 67–95.

[86] Kröger, T., Finkemeyer, B., Heuck, M., and Wahl, F. M. Compliant motion programming: The Task Frame Formalism revisited. *Journal of Robotics and Mechatronics 3* (2004), 1029–1034.

[87] Kronander, K. *Control and Learning of Compliant Manipulation Skills.* PhD thesis, 2015.

[88] Lagriffoul, F., and Benjamin, A. Combining task and motion planning: A culprit detection problem. *The International Journal of Robotics Research* (2016).

[89] Lagriffoul, F., Dimitrov, D., Bidot, J., Saffiotti, J., and Karlsson, L. Efficiently combining task and motion planning using geometric constraints. *The International Journal of Robotics Research 33*, 14 (2014), 1726–1747.

[90] Langley, P., and Choi, D. A unified cognitive architecture for physical agents. In *Proceedings of the National Conference on Artificial Intelligence* (2006), vol. 21, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, p. 1469.

[91] Leidner, D., Borst, C., and Hirzinger, G. Things are made for what they are: Solving manipulation tasks by using functional object classes. In *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on* (Nov 2012), pp. 429–435.

[92] Lemai, S., and Ingrand, F. Interleaving temporal planning and execution in robotics domains. In *Proceedings of the AAAI National Conference on Artificial Intelligence* (San Jose, California, USA, 2004), Citeseer.

[93] Levesque, H., and Lakemeyer, G. Cognitive robotics. Elsevier Science, 2008, pp. 869–886.

[94] LEVESQUE, H. J., REITER, R., LESPERANCE, Y., LIN, F., AND SCHERL, R. B. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming 31*, 1 (1997), 59–83.

[95] LUTSCHER, E., AND CHENG, G. Constrained manipulation in unstructured environment utilizing hierarchical task specification for indirect force controlled robots. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Hong Kong, 2014), pp. 3471–3476.

[96] MANSARD, N., AND CHAUMETTE, F. Task sequencing for sensor-based control. *IEEE Transactions on Robotics 23*, 1 (2007), 60–72.

[97] MANSARD, N., KHATIB, O., AND KHEDDAR, A. A unified approach to integrate unilateral constraints in the stack of tasks. *IEEE Transactions on Robotics 25*, 3 (2009), 670–685.

[98] MARZINOTTO, A., COLLEDANCHISE, M., SMITH, C., AND OGREN, P. Towards a unified behavior trees framework for robot control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Hong Kong, 2014), pp. 5420–5427.

[99] MASON, M. T. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 6 (1981), 418–432.

[100] MCCARTHY, J. Programs with common sense. In *Semantic Information Processing*, M. Minsky, Ed. 1968, pp. 403–418.

[101] MCCARTHY, J., AND HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence* (1969), 431–450.

[102] MCCARTHY, J. M., AND ROTH, B. Instantaneous properties of trajectories generated by planar, spherical, and spatial rigid body motions. *Transactions of the ASME, Journal of Mechanical Design 104* (1982), 39–51.

[103] MILLER, A., AND ALLEN, P. Graspit! a versatile simulator for robotic grasping. *Robotics Automation Magazine, IEEE 11*, 4 (Dec 2004), 110–122.

[104] MILLER, A., KNOOP, S., CHRISTENSEN, H., AND ALLEN, P. Automatic grasp planning using shape primitives. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on* (Sept 2003), vol. 2, pp. 1824–1829 vol.2.

[105] Minton, S., Bresina, J., and Drummond, M. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research 2* (1994), 227–262.

[106] Mösenlechner, L., and Beetz, M. Fast temporal projection using accurate physics-based geometric reasoning. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 1821–1827.

[107] Mudrova, L., and Hawes, N. Task scheduling for mobile robots using interval algebra. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Seattle, USA, 2015), pp. 383–388.

[108] Mueller, E. T. Event calculus and temporal action logics compared. *Artificial Intelligence 170*, 11 (2006), 1017–1029.

[109] Muñoz, P., R-Moreno, M., and Castaño, B. Integrating a pddl-based planner and a plexil-executor into the ptinto robot. In *Trends in Applied Intelligent Systems*, N. García-Pedrajas, F. Herrera, C. Fyfe, J. Benítez, and M. Ali, Eds., vol. 6096 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 72–81.

[110] Nakamura, Y., Hanafusa, H., and Yoshikawa, T. Task-priority based redundancy control of robot manipulators. *The International Journal of Robotics Research 6*, 2 (1987), 3–15.

[111] Nau, D., Ghallab, M., and Traverso, P. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[112] Nesnas, I. A. D., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., Madison, R., Guineau, J., McHenry, M., Shu, I.-H., and Apfelbaum, D. CLARAty: Challenges and steps toward reusable robotic software. *International Journal of Advanced Robotic Systems 3*, 1 (2006), 23–30.

[113] Object Management Group. Meta Object Facility (MOF) core specification. `http://www.omg.org/spec/MOF/2.4.1/PDF`, 2013.

[114] Pastor, P., Hoffmann, H., Asfour, T., and Schaal, S. Learning and generalization of motor skills by learning from demonstration. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (Kobe, Japan, 2009), ICRA2009, pp. 1293–1298.

[115] Paynter, H. M. An epistemic prehistory of Bond Graphs. In *Bond Graphs for Engineers*, P. Breedveld and G. Dauphin-Tanguy, Eds. 1992.

[116] Penrose, R. A generalized inverse for matrices. *Proc. Cambridge Philos. Soc. 51*, 3 (1955), 406–413.

[117] Perzylo, A., Somani, N., Profanter, S., Gaschler, A., Griffiths, S., Rickert, M., and Knoll, A. Ubiquitous semantics: Representing and exploiting knowledge, geometry, and language for cognitive robot systems. In *15th IEEE-RAS International Conference on Humanoid Robots* (Seoul, Republic of Korea, November 2015).

[118] Perzylo, A., Somani, N., Rickert, M., and Knoll, A. An ontology for CAD data and geometric constraints as a link between product models and semantic robot task descriptions. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Hamburg, Germany, 2015), IROS2015.

[119] Pettersson, O. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems 53* (2005), 73–88.

[120] Philippsen, R., Nejati, N., and Sentis, L. Bridging the gap between semantic planning and continuous control for mobile manipulation using a graph-based world representation. In *International Workshop on Hybrid Control of Autonomous Systems (HYCAS-09)* (2009), pp. 77–82.

[121] Philippsen, R., Sentis, L., and Khatib, O. An open source extensible software package to create whole-body compliant skills in personal mobile manipulators. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (San Francisco, California, 2011), IROS2011, pp. 1036–1041.

[122] Piao, Y., Hayakawa, K., and Sato, J. Space-time invariants for recognizing 3D motions from arbitrary viewpoints under perspective projection. *Transactions of the Institute of Electronics, Information and Communication Engineers E89-D*, 7 (2006), 2268–2274.

[123] Pnueli, A. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on* (Oct 1977), pp. 46–57.

[124] Prats, M., Pobil, A. P. D., and Sanz, P. J. *Robot Physical Interaction through the combination of Vision, Tactile and Force Feedback - Applications to Assistive Robotics*, vol. 84 of *Springer Tracts in Advanced Robotics*. Springer, 2013.

[125] Py, F., and Ingrand, F. Real-time execution control for autonomous systems. In *Embedded and Real-Time Systems* (2004), pp. 21–23.

[126] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T. B., Leibs, J., Wheeler, R., and Ng, A. Y. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software* (2009).

[127] Raatikainen, P. Gödel's incompleteness theorems. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., spring 2015 ed. 2015.

[128] RoboHow. The RoboHow Project. `http://robohow.eu/`.

[129] Rocchi, A., Hoffman, E., Caldwell, D., and Tsagarakis, N. Opensot: A whole-body control library for the compliant humanoid robot coman. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on* (Seattle, USA, 2015), pp. 6248–6253.

[130] ROSETTA. Robot control for skilled execution of tasks in natural interaction with humans; based on autonomy, cumulative knowledge and learning. `http://www.fp7rosetta.eu/`.

[131] Rusu, R. B., Marton, Z. C., Blodow, N., Dolha, M. E., and Beetz, M. Functional object mapping of kitchen environments. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on* (2008), IEEE, pp. 3525–3532.

[132] Rutgeerts, J. *Constraint-based task specification and estimation for sensor-based robot tasks in the presence of geometric uncertainty.* PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2007.

[133] Saab, L., Ramos, O. E., Keith, F., Mansard, N., Souères, P., and Fourquet, J.-Y. Dynamic whole-body motion generation under rigid contacts and other unilateral constraints. *IEEE Transactions on Robotics 29*, 2 (2013), 346–362.

[134] Sacerdoti, E. D. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1* (San Francisco, CA, USA, 1975), IJCAI'75, Morgan Kaufmann Publishers Inc., pp. 206–214.

[135] Samson, C., Le Borgne, M., and Espiau, B. *Robot Control, the Task Function Approach.* Clarendon Press, Oxford, England, 1991.

[136] Scioni, E., Borghesan, G., Bruyninckx, H., and Bonfè, M. A framework for formal specification of robotic constraint-based tasks and their concurrent execution with online qos monitoring. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Chicago, IL, USA, 2014), IROS2014, pp. 2963–2969.

[137] Scioni, E., Borghesan, G., Bruyninckx, H., and Bonfè, M. Bridging the gap between discrete symbolic planning and optimization-based robot control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Seattle, USA, 2015), pp. 5075–5081.

[138] Scioni, E., Hübel, N., Blumenthal, S., Shakhimardanov, A., Klotzbücher, M., Garcia, H., and Bruyninckx, H. Hierarchical hypergraphs for knowledge-centric robot systems: a composable structural meta model and its domain specific language npc4. *Journal of Software Engineering in Robotics* (under review).

[139] Scioni, E., Klotzbücher, M., De Laet, T., Bruyninckx, H., and Bonfé, M. Preview coordination: An enhanced execution model for online scheduling of mobile manipulation tasks. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Tokyo, Japan, 2013), IROS2013, pp. 5779–5786.

[140] Sentis, L., and Khatib, O. Task-oriented control of humanoid robots through prioritization. In *IEEE International Conference on Humanoid Robots* (2004).

[141] Sentis, L., and Khatib, O. A whole-body control framework for humanoid operating in human environments. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation* (Orlando, U.S.A., 2006), ICRA2006, pp. 2641–2648.

[142] Shakhimardanov, A. *Composable Robot Motion Stack: Implementing constrained hybrid dynamics using semantic models of kinematic chains.* PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, November 2015.

[143] Shakhimardanov, A., Bruyninckx, H., Copejans, M., and Smits, R. Popov-Vereshchagin algorithm for linear-time hybrid dynamics, control and monitoring with weighted or prioritized partial motion constraints in tree-structured kinematic chains. Under review., 2015.

[144] Shanahan, M. *Solving the frame problem: a mathematical investigation of the common sense law of inertia.* MIT Press, Cambridge, MA, 1997.

[145] Shanahan, M. The event calculus explained. In *Artificial intelligence today.* Springer, 1999, pp. 409–430.

[146] Shiraki, Y., Nagata, K., Yamanobe, N., Nakamura, A., Harada, K., Sato, D., and Nenchev, D. Modeling of everyday objects for semantic grasp. In *Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on* (2014), pp. 750–755.

[147] Simmons, R. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation 10*, 1 (1994), 34–43.

[148] Simmons, R., and Apfelbaum, D. A task description language for robot control. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vancouver, British Columbia, Canada, 1998), IROS98, pp. 1931–1937.

[149] Skubic, M. *Transferring assembly skills to robots: learning force sensory patterns and skills from human demonstration.* PhD thesis, Texas A&M University, 1997.

[150] Smits, R. *Robot skills: design of a constraint-based methodology and software support.* PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2010.

[151] Smits, R., Bruyninckx, H., and De Schutter, J. Software support for high-level specification, execution and estimation of event-driven, constraint-based multi-sensor robot tasks. In *Proceedings of the 2009 International Conference on Advanced Robotics* (Munich, Germany, 2009), ICAR2009.

[152] Smoljkic, G., Borghesan, G., Poorten, E. V., Reynaerts, D., and Sloten, J. V. *6th European Conference of the International Federation for Medical and Biological Engineering: MBEC 2014, 7-11 September 2014, Dubrovnik, Croatia.* Springer International Publishing, 2015, ch. Force Control of a Rigid Robot with a Flexible Link, pp. 375–378.

[153] Somani, N., Cai, C., Perzylo, A., Rickert, M., and Knoll, A. Object recognition using constraints from primitive shape matching. In *Proceedings of the International Symposium on Visual Computing* (Las Vegas, Nevada, USA, December 2014), ISVC2014, Springer, p. 783–792.

[154] Somani, N., Gaschler, A., Rickert, M., Perzylo, A., and Knoll, A. Constraint-based task programming with CAD semantics: From intuitive specification to real-time control. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Hamburg, Germany, 2015), IROS2015.

[155] Sporny, M., Longley, Dave Kellogg, G., Lanthaler, M., and Lindström, N. A JSON-based serialization for Linked Data. `http://www.w3.org/TR/json-ld/`, 2014.

[156] Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. Combined task and motion planning through an extensible planner-independent interface layer. In *Proceedings of the*

IEEE International Conference on Robotics and Automation (Hong Kong, May 2014), pp. 639–646.

[157] Tenorth, M., Bartels, G., and Beetz, M. Knowledge-based specification of robot motions. In *European Conference in Artificial Intelligence* (2014).

[158] Tenorth, M., and Beetz, M. KnowRob—Knowledge processing for autonomous personal robots. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (St. Louis, Missouri, 2009), IROS2009, pp. 4261–4266.

[159] Tenorth, M., and Beetz, M. KnowRob—A knowledge processing infrastructure for cognition-enabled robots. *The International Journal of Robotics Research 32*, 5 (2013), 566–590.

[160] Tenorth, M., Perzylo, A. C., Lafrenz, R., and Beetz, M. Representation and exchange of knowledge about actions, objects, and environments in the ROBOEARTH framework. *IEEE Transactions on Automation Science and Engineering* (2013).

[161] Thomas, U., Finkemeyer, B., Kröger, T., and Wahl, F. M. Error-tolerant execution of complex robot tasks based on skill primitives. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation* (Taipeh, Taiwan, 2003), ICRA2003, pp. 3069–3075.

[162] Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., and Wortmann, A. A new skill based robot programming language using uml/p statecharts. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 461–466.

[163] Ureche, A., Umezawa, K., Nakamura, Y., and Billard, A. Task parameterization using continuous constraints extracted from human demonstrations. *IEEE Transactions on Robotics 31*, 6 (Dec 2015), 1458–1471.

[164] van de Molengraft, M. The RoboEarth project. `http://www.roboearth.org/`, 2011.

[165] Vanthienen, D. *Composition Pattern for Constraint-based Programming with Application to Force-sensorless Robot Tasks.* PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, January 2015.

[166] Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., and De Schutter, J. Force-sensorless and bimanual human-robot comanipulation. In *10th IFAC Symposium on Robot Control (SYROCO)* (Dubrovnik, Croatia, September, 5–7 2012), vol. 10.

[167] Vanthienen, D., Klotzbücher, M., and Bruyninckx, H. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *Journal of Software Engineering in Robotics 5*, 1 (2014), 17–35.

[168] Vanthienen, D., Klotzbücher, M., De Laet, T., De Schutter, J., and Bruyninckx, H. Rapid application development of constrained-based task modelling and execution using domain specific languages. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Tokyo, Japan, 2013), IROS2013, pp. 1860–1866.

[169] Veloso, M. Graph-based task libraries for robots: Generalization and autocompletion. In *AI* IA 2015 Advances in Artificial Intelligence: XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, September 23-25, 2015, Proceedings* (2015), vol. 9336, Springer, p. 397.

[170] Verma, V., Estlin, T., Jónsson, A., Pasareanu, C., Simmons, R., and Tso, K. Plan execution interchange language (plexil) for executable plans and command sequences. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space* (2005).

[171] Verscheure, D., Demeulenaere, B., Swevers, J., Schutter, J. D., and Diehl, M. Time-optimal path tracking for robots: A convex optimization approach. *IEEE Transactions on Automatic Control 54*, 10 (2009), 2318–2327.

[172] Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., and Carrasco, R. C. Probabilistic finite-state machines—Part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence 27*, 7 (2005), 1013–1025.

[173] Winkler, J., Bartels, G., Mösenlechner, L., and Beetz, M. Knowledge Enabled High-Level Task Abstraction and Execution. *First Annual Conference on Advances in Cognitive Systems 2*, 1 (December 2012), 131–148.

List of publications

- Enea Scioni, Gianni Borghesan, Herman Bruyninckx, Marcello Bonfè.
  Bridging the gap between Discrete Symbolic Planning and Optimization-based Robot Control.
  In: *Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, USA.

- Gianni Borghesan, Enea Scioni, Abderrahmane Kheddar, Herman Bruyninck.
  Introducing Geometric Constraint Expressions into Robot Constrained Motion Specification and Control.
  In: *IEEE Robotics and Automation Letters, Volume:PP, Issue: 99, Year: 2015, doi: 10.1109/LRA.2015.2506119* .

- Enea Scioni, Nico Hüebel, Sebastian Blumenthal, Azamat Shakhimardanov, Markus Klotzbücher, Hugo Garcia, Herman Bruyninckx.
  Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain-Specific Language NPC4. In: *accepted to Journal of Software Engineering for Robotics (JOSER), 2015.*

- Enea Scioni, Gianni Borghesan, Herman Bruyninckx, Marcello Bonfè.
  A Framework for Formal Specification of Robotic Constraint-based Tasks and their Concurrent Execution with Online QoS Monitoring.
  In: *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Chicago, USA.

- Enea Scioni, Markus Klotzbüecher, Tinne De Laet, Herman Bruyninck, Marcello Bonfè.
  Preview coordination: An enhanced execution model for online scheduling of mobile manipulation tasks.
  In: *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan.

- Marcello Bonfè, Enea Scioni, Cristian Secchi.
  Online Trajectory Generation and Tracking Control Design for Mobile Robots with Kinodynamic Constraints.
  In: *Proceedings of the 10th IFAC Symposium on Robot Control (SYROCO 2012).*

- Koen Buys, Steven Bellens, Wilm Decre, Ruben Smits, Enea Scioni, Tinne De Laet, Joris De Schutter, Herman Bruyninckx.
  Haptic coupling with augmented feedback between two KUKA Light-Weight Robots and the PR2 robot arms.
  In: *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, USA.