Systematic use of models of concurrency in executable domain-specific modelling languages

Florent Latombe

▶ **To cite this version:**

Florent Latombe. Systematic use of models of concurrency in executable domain-specific modelling languages. Other [cs.OH]. Institut National Polytechnique de Toulouse - INPT, 2016. English. NNT : 2016INPT0057 . tel-04247547v2

HAL Id: tel-04247547

https://theses.hal.science/tel-04247547v2

Submitted on 18 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Sureté de Logiciel et Calcul à Haute Performance

Présentée et soutenue par :

M. FLORENT LATOMBE

le mercredi 13 juillet 2016

Titre :

SYSTEMATIC USE OF MODELS OF CONCURRENCY IN EXECUTABLE DOMAIN-SPECIFIC MODELING LANGUAGES

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. YAMINE AIT AMEUR

M. XAVIER CREGUT

Rapporteurs :

M. ANTONIO VALECILLO, UNIVERSIDAD DE MALAGA

M. RICHARD PAIGE, UNIVERSITY OF YORK

Membre(s) du jury :

M. FREDERIC BOULANGER, SUPELEC, Président

M. JULIEN DE ANTONI, INRIA SOPHIA ANTIPOLIS, Membre

M. MARC PANTEL, INP TOULOUSE, Membre

M. XAVIER CREGUT, INP TOULOUSE, Membre

All that is gold does not glitter,
Not all those who wander are lost;
The old that is strong does not wither,
Deep roots are not reached by the frost.
From the ashes a fire shall be woken,
A light from the shadows shall spring;
Renewed shall be blade that was broken,
The crownless again shall be king.

in *The Fellowship of the Ring*,
by J. R. R. Tolkien (1892 – 1973).

Acknowledgements

This too short of a section is dedicated to my acknowledging of all the people who have shared their experience, mentorship, friendship, kindness and love with me during these past few years.

I can't thank enough my supervisors Marc Pantel and Xavier Crégut for giving me the opportunity to do a PhD under their guidance. Thanks for trusting me.

I also want to thank all the active members of the ANR INS Project GEMOC whose influence on this thesis I can never come close to quantify correctly. Thanks for all the discussions and advices you have given me.

I can't skip over thanking all the people I have met while working at IRIT ; researchers, secretaries, and fellow doctoral students alike.

I want to thank the supervisors during my stay at the National Institute of Informatics in Tokyo, Soichiro Hidaka and Zhenjiang Hu. I have learned a lot from you over too short a time.

I would like to thank my family for supporting me and for not making me have to explain the topic of my PhD too many times.

Thanks to all my friends in Toulouse and elsewhere for sticking around. I apologize if I have encouraged any of you to start your own PhD.

Finally, 'Thanks' would not even begin to cover my gratitude for the support I get at home. *Tusen tack!*

This thesis has been redacted using the TeXstudio editor[1]*. The template used is a modification of Krishna Kumar's thesis template for the Cambridge University Engineering Department*[2]*.*

[1]http://www.texstudio.org/

[2]https://github.com/kks32/phd-thesis-template

Abstract

The complexity of modern softwares and systems, like the Internet of Things or Cyber-Physical Systems, has been increasing regularly since the birth of computing. They integrate many features, possibly relying on a variety of networks or other systems; comply to different norms, including security and safety standards; all the while reacting in a timely manner. Their design and development is costly, both in the required engineering effort, and possibly also in terms of their raw physical parts. Their updating and maintenance processes are also complex to handle. To ease these activities, researchers in software engineering have proposed new development paradigms. In this context, *Language-Oriented Programming* (LOP) proposes to make languages first-class citizens in the software engineering activities. LOP advocates using multiple Domain-Specific Languages (DSLs), each specialized for a particular problem domain. Since modern systems are usually designed by domain experts, the use of Domain-Specific Modeling Languages (DSMLs) is gaining traction, because their abstractions are designed to be intuitive for the end users, *i.e.*, the domain experts. This has led to the development of a new discipline called *Software Language Engineering*, which studies the design, implementation and tooling of Domain-Specific (Modeling) Languages. To facilitate the early verification and validation activities of these systems, DSMLs can be made eXecutable (xDSMLs). In the context of Model-Driven Engineering (MDE), the design of xDSMLs has led to the development of several "Executable Metamodeling" approaches, where models are executable according to an execution semantics defined at the metamodel (abstract syntax) level.

Modern softwares and systems are also increasingly concurrent, to accommodate for their increasing scale in users, features, and overall importance in our societies. To ensure an adequate behavior, notably in terms of their interfacing with users or with other systems, or simply in terms of performance, their execution environments provide more and more parallel facilities, such as GPGPU pipelines, many-core CPUs or FPGAs. To facilitate their deployment on various platforms, highly-concurrent softwares must be developed without prior knowledge of their execution platforms, while still allowing full exploitation of the available parallel facilities at runtime. The specification of the concur-

rency concerns of modern systems is thus placed at the heart of the software engineering activities. Theoretical computer science has studied several paradigms for this purpose, commonly denominated as Models of Concurrency (MoCs). MoCs are formalisms which are practical for the analytical study of properties relating to concurrency, such as detecting deadlocks, starvation situations or liveness properties of critical parts of a system. But using a MoC is complex, since it requires both theoretical knowledge about the MoC, practical knowledge about its use, and overall solid knowledge of the system being developed. Even if explicitly used, it is essentially hard-coded for a specific system, and little to no guarantee of its correct use is ensured by the language.

We build upon an existing xDSML design approach, published in the International Conference on Software Language Engineering 2012 [18] and 2013 [19], which attempts to bridge the gap between LOP and MoCs, by designing so-called *Concurrency-aware xDSMLs*. In these languages, the concurrency concerns of an xDSML are made explicit using a dedicated formalism based on a MoC. By making these concerns explicit at the language level, the correct use of a MoC for any program conforming to the syntax of the language is ensured. They can be specialized at design time, to implement a particular Semanptic Variation Point (SVP) of the language, or refined at deployment time, for a specific execution platform. Consequently, the concurrency aspects of a system can be analyzed, via model-checking tools for instance.

In this thesis, we detail and improve upon the design of concurrency-aware xDSMLs. We first focus on the separation of concerns inside the operational semantics specification. The concurrency concerns are separated from the data and functional operations aspects of the semantics. Executing a model is done by coordinating the execution of these two concerns. This coordination is specified thanks to a third concern, making explicit the communication between the first two concerns. We study the possible coordinations that can be specified, and how they are realized at runtime. Then, we focus on the Model of Concurrency used in the initial approach: Event Structures. This formalism is not a good fit for all xDSMLs, thus we propose a recursive definition of concurrency-aware xDSMLs, enabling the use of any concurrency-aware xDSML as a MoC. This approach provides a formal definition and interface for MoCs, as well as allows xDSMLs to use an adapted language for the specification of their concurrency. Finally, we step away from *operational* semantics and propose an approach to define the semantics of concurrency-aware xDSMLs in a *translational* manner, based on any previously-defined concurrency-aware xDSML. We detail the advantages and drawbacks of using translational semantics instead of operational semantics in the context of concurrency-aware xDSMLs, and propose an approach to catch up on some of the execution facilities provided by the operational semantics approach.

À l'attention des lecteurs francophones

To the French-speaking Readers

Conformément à l'Article L121-3 du Code de l'Éducation[3] ; à la décision du Conseil Scientifique de l'Université Toulouse III Paul Sabatier[4] ; et aux directives de l'École Doctorale 475 Mathématiques, Informatique et Télécommunications de Toulouse[5] ; cette thèse est rédigée en language anglaise suite à l'obtention d'une autorisation adéquate. Des résumés substantiels, en langue française, de la thèse, de chaque chapitre ainsi que des annexes sont inclus tout le long du document. Ci-dessous, le sommaire exhaustif de ces résumés.

Sommaire des résumés en language française

---

[3] https://www.legifrance.gouv.fr/affichCodeArticle.do?idArticle=LEGIARTI000027747711&cidTexte=LEGITEXT000006071191

[4] http://www.univ-tlse3.fr/conseil-scientifique-/

[5] http://www.edmitt.ups-tlse.fr/

Résumé

Abstract (in French)

La complexité des systèmes et logiciels contemporains, tels que l'internet des objets (*the Internet of Things*) ou les systèmes cyber-physiques (*Cyber-Physical Systems*), ne cesse de croître. Le nombre de fonctionnalités et d'intéractions qu'ils doivent gérer s'élargit sans arrêt, et celles-ci impliquent généralement un nombre croissant d'acteurs tel des réseaux de différentes natures, ainsi que d'autres systèmes ou logiciels. Ils doivent aussi respecter de nombreuses normes de sûreté et de sécurité, tout en fonctionnant suivant un temps de réponse acceptable pour les acteurs externes (utilisateurs ou autres systèmes). Leur conception et développement sont de plus en plus coûteux, principalement en terme d'ingénierie, mais aussi possiblement en terme de composants matériels. Leurs processus de mises à jour et de maintenance deviennent eux aussi, en conséquence, plus complexes que jamais.

Afin de faciliter ces différentes activités, les chercheurs en génie du logiciel doivent proposer de nouveaux paradigmes de développement. C'est dans ce contexte qu'a été proposée la programmation orientée langages (*Language-Oriented Programming – LOP*). Cette approche place l'utilisation de langages informatiques adéquates au centre des activités d'ingénierie. Plus précisément, elle repose sur la conception et l'utilisation de nombreux langages dédiés (*Domain-Specific Languages – DSLs*) différents, chacun d'entre eux étant dédié à l'expression de la solution d'un aspect particulier du système. Les systèmes complexes étant le plus souvent conçus par des experts métiers (circuits électriques, circuits hydrauliques, mécanique des fluides, réseaux, sécurité, etc.), les langages de modélisation dédiés (*Domain-Specific Modeling Languages – DSMLs*) sont plus adaptés car leurs concepts sont précisément faits de façon à correspondre à un domaine métier. Ceci facilite l'utilisation de langages informatiques par les experts métiers, experts qui ne sont pas nécessairement formés à la programmation informatique. Ce paradigme a donné naissance à une discipline appelée *Ingénierie des Langages* (*Software Language Engineering*),

qui se focalise sur la conception, l'implémentation et l'outillage des langages (de modélisation) dédiés.

Lors de la conception d'un système, la possibilité de pouvoir simuler son comportement permet d'effectuer des activités de vérification et de validation de ce système. Cela peut permettre de valider une spécification, de détecter des erreurs de conception, ou de réaliser des étapes de validation intermédiaires dès le début du processus d'ingénierie. Les DSMLs permettant cela sont dits "exécutables" (*eXecutable DSMLs – xDSMLs*). Dans le cadre de l'Ingénierie Dirigée par les Modèles (*Model-Driven Engineering – MDE*), la popularité des *xDSMLs* a conduit au développement de nombreuses approches dites de "métamodélisation exécutable" (*Executable Metamodeling*). Un modèle conforme à un métamodèle est exécutable selon une sémantique d'exécution définie au niveau du métamodèle, qui représente alors la syntaxe abstraite du langage.

Les systèmes et logiciels complexes sont aussi de plus en plus concurrents ; conséquence de leur complexité et du passage à l'échelle en termes d'utilisateurs et de fonctionnalités à gérer. Pour que leur exécution demeure adéquate, notamment dans le contexte d'une interface avec des utilisateurs ou d'autres systèmes, ou tout simplement pour améliorer leur performance (rapidité d'exécution, temps de réponse, etc.), les plateformes sur lesquelles ils s'exécutent sont dotées de capacités de parallélisation telles que des processeurs graphiques (*GPGPU*), des processeurs multi-cœurs (*many-core CPUs*) ou des réseaux de portes programmables *in situ* (*Field-Programmable Gate Arrays – FPGAs*). Afin de permettre leur déploiement sur des plateformes de natures diverses, ces systèmes doivent être développés sans connaissance préalable de la plateforme d'exécution finale, tout en étant spécifié de façon à pouvoir bénéficier d'éventuelles capacités de parallélisation. La spécification correcte des aspects concurrents de ces systèmes est donc au cœur du développement logiciel.

Dans le domaine de la recherche en informatique théorique, plusieurs formalismes ont été développés dans le but de spécifier les aspects concurrents d'un système. Ces formalismes sont appelés modèles de concurrence (*Models of Concurrency – MoCs*). Ils permettent l'étude analytique de propriétés liées aux aspects concurrents d'un système tels que la détection de situations d'interblocage, de famine, etc. Cependant, l'utilisation d'un *MoC* est complexe : elle nécessite une bonne connaissance théorique du *MoC*, un savoir-faire relatif à son implémentation et à son utilisation, ainsi qu'une expertise du comportement du système que l'on cherche à spécifier. L'utilisation d'un *MoC* est donc souvent restreinte à un système donné, et peu de garanties sur la correction de son utilisation peuvent être assurées.

Nos travaux visent à combiner l'approche *LOP* avec l'utilisation de *MoCs*. Ils reposent sur une approche existante, initialement publiée dans l'*International Conference on Soft-*

ware Language Engineering 2012 [18] et 2013 [19], qui pose les bases de la conception de *xDSMLs* pour lesquels les aspects concurrents de la sémantique d'exécution sont explicités à l'aide de l'utilisation d'un *MoC* (*Concurrency-aware xDSMLs*). L'utilisation d'un *MoC* est spécifiée au niveau du langage, et non du système, sur la base d'un *MoC* existant. Cette approche garantit l'utilisation cohérente d'un *MoC* par tous les modèles conformes au même *xDSML*. Des outils dédiés à la vérification de modèles (*model-checking*) peuvent ensuite être appliqués sur les aspects concurrents spécifiques à un modèle. Ces langages avec concurrence explicite peuvent aussi être raffinés, par exemple afin d'implémenter un point de variation sémantique (*Semantic Variation Point*) du langage, ou bien pour le spécialiser à une plateforme d'exécution particulière.

Dans cette thèse, nous détaillons et améliorons la conception de *concurrency-aware xDSMLs*, et l'exécution de modèles conformes à ces langages. Dans un premier temps, nous nous concentrons sur la séparation des préoccupations au sein de la sémantique opérationnelle. Nous séparons les aspects concurrents d'une part, des aspects liés aux données et à leur évolution d'autre part. L'exécution d'un modèle est ensuite réalisée par la coordination de ces deux préoccupations. Cette coordination est définie à l'aide d'un troisième élément représentant la *communication* entre ces deux aspects. On détaillera, notamment, les différentes formes de coordination qui peuvent être définies. Cette approche repose dans un premier temps sur l'utilisation d'un *MoC* particulier : les structures d'évènements (*Event Structures*). Les *MoCs* correspondant à un paradigme de concurrence particulier, ils sont plus ou moins adaptés pour un domaine (et par extension, pour un langage) donné. Nous proposons donc une approche permettant la définition et l'utilisation de nouveaux *MoCs*. Notre proposition repose sur une définition récursive de la spécification des *concurrency-aware xDSMLs*, dans laquelle le *MoC* est un *concurrency-aware xDSML* défini préalablement. Enfin, nous proposons une approche *translationnelle* de la sémantique de ces langages, c'est-à-dire une définition de la sémantique d'exécution d'un langage reposant entièrement sur l'utilisation d'un autre *concurrency-aware xDSML*. Cette approche facilite la spécification de nouveaux langages, mais certains avantages de l'utilisation de *concurrency-aware xDSMLs* sont perdus par l'utilisation de cette technique. Nous proposons donc une solution permettant de pallier ces inconvénients.

Table of Contents

List of Figures

List of Source Code Listings

"I must not fear. Fear is the mind-killer. Fear is the little-death that brings total obliteration. I will face my fear. I will permit it to pass over me and through me. And when it has gone past I will turn the inner eye to see its path. Where the fear has gone there will be nothing. Only I will remain."

Litany against fear, in *Dune*, by Frank Herbert (1920 – 1986).

1

Introduction and Objectives

Summary

We summarize the context of our work: modern, highly-concurrent, software-intensive systems, deployed and executed on increasingly-parallel platforms. We then introduce two research fields that we bring together in this thesis: the Language-Oriented Programming (LOP) paradigm, concretized through the design and implementation of eXecutable Domain-Specific Modeling Languages; and Models of Concurrency (MoCs), used to provide high-level concurrency constructs to computer languages. We present the research context within which this thesis was realized, and the objectives of this thesis with regards to LOP and MoCs. Finally, we lay out the organization of the rest of this document.

Chapter Outline

Résumé

Nous présentons dans ce chapitre les contextes scientifique et académique de nos travaux, ainsi que les objectifs de notre thèse.

L'omniprésence des ordinateurs dans notre vie quotidienne a fait du génie logiciel une discipline phare de notre société moderne. Combinées à la complexité exponentielle des logiciels qu'ils exécutent, les activités de conception, développement, mise au point, test, refactorisation, simulation et exécution d'un logiciel sont plus complexes que jamais. De nouveaux paradigmes de génie logiciel doivent donc être développés, outillés et enseignés. Les paradigmes à base de modèles, tels que l'ingénierie à base de modèles (*Model-Based Software Engineering – MBSE*) dans lesquels les modèles sont un concept clé, dérivant jusqu'à l'ingénierie dirigée par les modèles (*Model-Driven Engineering – MDE*) lorsque ceux-ci représentent le cœur même du processus, ont prouvé leur efficacité dans les industries qui les ont adoptés.

Cependant, le coût de cette adoption demeure élevé. Les langages de modélisation généralistes comme *UML* (*Unified Modeling Language*) peuvent être utilisés pour de nombreux domaines et de nombreuses activités, mais nécessitent des investissements en termes d'outils, d'infrastructures et de formations qui peuvent être prohibitifs. La généricité de ces langages est au prix de leur complexité, et du coût qui en découle. Des problèmes similaires se retrouvent dans les langages de programmation, pour lesquels de nombreux *frameworks* et bibliothèque sont développés afin de permettre la résolution de problèmes particuliers. L'utilisation de ces outils devient plus complexe à mesure que les problèmes s'intensifient. Pour pallier cela, il est possible de créer des langages dédiés (*Domain-Specific Languages – DSLs*) qui se focalisent sur la résolution d'une classe de problèmes donnée, à l'aide d'une syntaxe et d'une sémantique d'exécution adaptée. Dans l'approche MBSE, ils se concrétisent sous la forme de langages de modélisation dédiés (*Domain-Specific Modeling Languages – DSMLs*), autrement dit, des langages adaptés à la résolution des problèmes d'un domaine métier particulier, représenté sous forme de métamodèle, et dont les solutions peuvent être formulées par des experts du domaine (éventuellement ignorants des technologies liées à la programmation).

Ces langages sont dits exécutables (*xDSMLs*) lorsqu'ils disposent d'une sémantique d'exécution. Les programmes conformes à un *xDSML* (qui sont donc des modèles de systèmes) peuvent être exécutés, c'est-à-dire que leur chargement par un environnement d'exécution comme un système d'exploitation ou une machine virtuelle conduit à une simulation du système réel représenté par le programme. Ceci permet de vérifier et de valider le comportement du système très tôt dans le processus de développement logiciel.

Le développement de *xDSMLs* est au cœur d'une approche appelée programmation orientée langages (*Language-Oriented Programming – LOP*). La création et l'outillage de *xDSMLs* demeure complexe et réservée à des experts en théorie des langages informatiques et technologies associées. En particulier, la définition de la sémantique d'exécution peut très rapidement devenir extrêmement complexe pour des langages avec un haut niveau de concurrence. Or, les *xDSMLs* doivent permettre la spécification des systèmes complexes, qui sont souvent extrêmement concurrents, et/ou exécutés à l'aide de plateformes concurrentes (distribuées, hautement parallèles, etc.). Les techniques actuelles de développement de *xDSMLs* rendent difficile la définition des aspects concurrents d'un langage indépendamment de toute plateforme d'exécution particulière. En conséquence, les aspects concurrents d'un *xDSML* émanent soit implicitement de la plateforme d'exécution utilisée, ou bien de l'implémentation du langage exploité. Langages et systèmes sont donc difficiles à raffiner, par exemple pour passer d'une plateforme séquentielle à une plateforme hautement parallèle. Si tant est qu'il soit possible, ce raffinement est, pour un système, fait le plus souvent de façon manuelle. Ceci nécessite de bien connaître le modèle de concurrence (*Model of Concurrency – MoC*) utilisé (réseaux de Pétri – *Petri nets* ; structures d'évènements – *Event Structures* ; modèle d'acteur – *Actor model* ; etc.). Dans les langages de programmation généralistes, ces *MoCs* sont le plus souvent très génériques (permettant de les utiliser pour tous types de système) et accessibles à l'aide d'un *framework* ou d'une bibliothèque, ce qui permet de les combiner librement. L'utilisation correcte d'un *MoC* doit donc être assurée par le concepteur du système, qui doit donc être formé à l'utilisation à la fois du langage utilisé et du *MoC* choisi, tout en étant un expert du système.

Dans cette thèse, nous souhaitons faciliter la spécification des aspects concurrents des systèmes et des langages. Nous formalisons et étendons une approche permettant la définition de *xDSMLs* dans lesquels les aspects concurrents sont explicités à l'aide d'un métalangage adapté, sur la base d'un *MoC*. L'approche permet aussi l'exécution des programmes conformes à ces langages. Ces *xDSMLs* sont dits *concurrency-aware*, car dans la sémantique d'exécution les aspects concurrents sont explicites, au contraire des approches traditionnelles dans lesquelles ils sont généralement diffus, et donc difficiles à identifier, analyser et raffiner. La définition de ces langages repose sur la spécification de comment un *MoC* est utilisé de façon systématique pour tout modèle conforme au langage. Cette spécification peut ensuite être raffinée pour particulariser le langage à une plateforme d'exécution spécifique. L'utilisation du *MoC* pour un modèle peut aussi être utilisée pour des analyses telles que la recherche d'interblocages ou de famines, permettant de garantir la validité du comportement concurrent du système. Enfin, cette approche facilite grandement la spécifi-

cation de systèmes complexes, puisque l'utilisation d'un *MoC* est faite de façon mécanique, grâce à son intégration au niveau du langage.

Cette thèse a été réalisée dans le cadre du projet *ANR INS GEMOC*, qui étudie les problématiques de la définition des aspects concurrents de la sémantique d'exécution des *xDSMLs*, des interfaces structurelles et comportementales des *xDSMLs*, et de la coordination entre *xDSMLs*. Ce projet regroupe INRIA (Institut National de Recherche en Informatique et en Automatique) à travers l'IRISA[1] (Institut de Recherche en Informatique et Systèmes Aléatoires) de Rennes et I3S[2] (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis) de Sophia Antipolis ; Thales Research & Technology[3] ; Obeo[4] ; l'ENSTA Bretagne[5] (École Nationale Supérieure de Techniques Avancées de Bretagne) et l'IRIT[6] (Institut de Recherche en Informatique de Toulouse).

Dans le Chapitre 2 nous présentons les travaux qui ont servi de fondation à cette thèse. En particulier, nous définissons la notion de concurrence et de modèles de concurrence. Nous présentons les approches traditionnelles de définition de langages et de leur sémantique d'exécution, et les moins traditionnelles, c'est-à-dire à base de modèles dans le cadre de l'Ingénierie Dirigée par les Modèles (*Model-Driven Engineering – MDE*). Dans le Chapitre 3, nous présentons comment définir la sémantique d'exécution d'un *concurrency-aware xDSML*. Nous analysons ensuite les limites de l'approche, par exemple certaines constructions de langage sont complexes ou impossibles à spécifier, et proposons des solutions. Nous présentons aussi l'implémentation dans un atelier de langage basé sur la plateforme Eclipse. Dans le Chapitre 4, nous étudions l'intégration de nouveaux *MoCs* dans l'approche, qui est initialement limitée au *MoC* des structures d'évènements. Cette contribution repose sur une définition récursive des *concurrency-aware xDSMLs*, permettant l'utilisation, en tant que *MoC*, d'un *concurrency-aware xDSML* précédemment défini. Enfin, nous étudions dans le Chapitre 5, comment définir la sémantique d'exécution de façon translationnelle, tout en ayant une spécification explicite des aspects concurrents de cette sémantique. Le Chapitre 6 présente la conclusion et les perspectives de nos travaux de thèse.

[1] http://diverse.irisa.fr/
[2] https://team.inria.fr/aoste/
[3] https://www.thalesgroup.com/fr/worldwide/global-innovation/recherche-technologie
[4] https://www.obeo.fr/fr/
[5] http://www.ensta-bretagne.fr/stic/index.php/ingenierie-dirigees-par-les-modeles/
[6] https://www.irit.fr/-ACADIE-team-

1.1 Context

1.1.1 Technological Context

Computers are everywhere. Their pervasiveness in our daily lives has made software engineering a key discipline in modern societies. Not only are computers more vital than ever, but the complexity of the software they host is also higher than ever. Designing, developing, debugging, testing, refactoring, simulating, implementing and executing software systems has never been more challenging. These challenges are even more prevalent for the highly-concurrent, highly-distributed, fault-tolerant systems of tomorrow: the Internet of Things, Cyber-Physical Systems, Smart Grids and Cities, etc.

To address these problems, suitable programming paradigms must be designed, tooled, and taught. In Model-Based Software Engineering (MBSE), models are important artefacts used for the formulation of the architecture, conception, deployment, behavior, etc. of a system. In Model-Driven Engineering (MDE), models are *the* key artefacts of the engineering activities. MBSE and MDE have proven effective at efficiently capturing the complexity of modern software-intensive systems [78], thus facilitating their development and maintenance.

Yet, these paradigms still come at a heavy cost. General-purpose Modeling Languages (GMLs) like the Unified Modeling Language (UML) [111] offer generic constructs for the specification of systems, but require sophisticated tooling, adequate training, and their genericity usually complicates the specification of key business solutions. Moreover, modern systems are usually designed by domain experts, who do not necessarily have a background in software engineering, modeling or even a computer-related field. This is a heavy weight against the general adoption of GMLs as a suitable paradigm. In fact, a GML often consists of several sub-languages, integrated together to form the GML. For instance, UML is composed of different diagrams (Class Diagram, Object Diagram, Package Diagram, Component Diagram, Activity Diagram, State Machine Diagram, Sequence Diagram, etc.), each with its own syntax and semantics. A similar issue is found in the programming community: scientists need to integrate their computations with tools or frameworks based on General-purpose Programming Languages (GPLs) such as Java, C++ or Python; database administrators need to allow applications written in GPLs to interact with their databases; front-end designers need practical constructs for the presentation of data, etc. GPLs are usually equipped with libraries providing the tools to realize certain tasks. Although they are effectively written using the same language, different libraries of a same GPL may have extreme differences in their syntactic (*e.g.*, naming, types used, nature of exceptions

thrown, etc.) and semantic (*e.g.*, free of side-effects, or relying on them, optimized for a particular hosting platform, etc.) aspects.

These problematics have lead to the development of *Domain-Specific Languages* (DSLs). DSLs provide programmers with powerful abstractions, facilitating the specification of a particular solution. In MBSE, they are concretized as *Domain-Specific Modeling Languages* (DSMLs): languages whose constructs and semantics are focused on a particular problem domain, and whose abstractions are intuitive for the domain experts to formulate solutions in. In these languages, the "programs" are models conforming to a metamodel (the data model, or abstract syntax, of the language), representing a real-world system (or relevant parts of it). The distinction between DSLs and general-purpose langages is blurry; even among GPLs, arguments can be made for using one or the other (*e.g.*, C or C++ for embedded devices due to its closeness to hardware languages, Python for scientific computing due to its numerous libraries and ease-of-use, Java for its cross-platform interoperability and the JVM ecosystem, etc.). It is not uncommon for softwares to combine several GPLs simply because some parts are more adequately addressed by some particular GPL. DSLs simply stretch this principle to the point where they are more adequate for a specific class of problem (*i.e.*, its *domain*), and where they often abandon some of the general-purpose features because they are not needed for the addressed domain. In that sense, DSLs are thus often considered as "simpler" languages than GPLs. This makes them, by construction, the "right tool for job", provided the domain is adequate for the problem at hand.

By defining an *execution semantics* (also called dynamic semantics or behavioral semantics) for a DSML, it can be made eXecutable (xDSML). Models conforming to an xDSML are executable, that is, when one is loaded by the execution environment of the language (*i.e.*, an operating system or a virtual machine), it produces a simulation of the real-world system represented by the model. Executability enables the early verification and validation of the systems being designed, *i.e.*, there is no need to reach the deployment phase of the system in order to ensure its behavior is as expected. This saves a lot of time (for the system designers and domain experts) and associated costs (hardware, support, etc.). Ultimately, MDE advocates that the real-world software system be generated based on its models, therefore guaranteeing the conformity of the system to its model. This code generation stage, however, raises its own set of challenges.

Designing languages, and even more so, xDSMLs, is complex. Existing language design approaches usually focus on the syntactic aspects of a language (*i.e.*, its concepts, how they are represented, and how they can be manipulated by the end user) and its tooling (*i.e.*, editor features, possibly integrated into an IDE), relying often on ad-hoc solutions for the semantical aspects. The main challenges remaining in defining xDSMLs thus en-

tail the specification of execution semantics, and the definition of the associated tooling (interpreter, compiler, animator, debugger, etc.).

To enable the specification of highly concurrent and distributed softwares, xDSMLs must have a rich concurrent semantics, allowing the use of modern highly-parallel platforms such as GPGPU pipelines, multi-core CPUs, distributed networks, etc. The parallel capacities of the platform(s) should not leak into the system design, but instead be abstracted away and dealt with in the deployment phase. However, current language development techniques make this difficult. Systems are often designed with a specific execution platform (or a family of platforms) in mind, and so are languages. In particular xDSMLs often do not make explicit which concurrency model they use, relying instead on the implementation or on a specific platform to provide one, thus preventing its analysis, variation and refinement. Moreover, the specialization to a specific execution platform (*e.g.*, distributed, sequential, highly-parallel, etc.) is usually given at the program level, but this activity requires specific knowledge about the theoretical model involved. These theoretical models are known as "Models of Concurrency" (MoCs). Notable MoCs in the literature include Petri nets [107], Event Structures [160] and the Actor model [65]. In GPLs, MoCs can be used through language constructs, libraries or frameworks, but their use is complex and submitted to many implicit rules. The mapping from the concurrency-related language constructs towards an execution platform is typically hard-coded (thus placing emphasis on which implementation of the language to use), or relies on an underlying execution platform (deferring these decisions to another component whose implementation may matter). For instance, Python applications behave very differently depending on the implementation used: the C implementation (CPython) is subject to the Global Interpreter Lock (GIL), preventing concurrent threads from executing in parallel ; while the Java implementation (Jython) uses the threads of the Java Virtual Machine (JVM). Since Java 1.3, most JVM implementations bind these to kernel threads, which can thus be executed in parallel on multi-core CPUs. However, that is an arbitrary implementation choice made by the JVM used. The Python specification allows both versions of threading, but some programs will execute poorly with one or the other interpreter (*e.g.*, a computation-heavy program may exploit the parallel capacities of Jython, while taking too much time when executed using CPython; a program with a lot of non-blocking operations such as input/output interactions may execute poorly with Jython due to the cost of context switching between threads), unless specifically adapted for it, which ties the program to a specific platform.

In this thesis, we focus on integrating the use of MoCs in the definition of the execution semantics of xDSMLs. We argue that the domain-specificity of xDSMLs allows them to not only capture domain-related meanings in the semantics, but also domain-related concur-

rency concerns. Our approach enables the refinement of xDSMLs for specific execution platforms, by specializing a part of the execution semantics during the deployment of the language. The concurrency concerns pertaining to a specific model can also be analyzed depending on the MoC on which it is based. Finally it also protects the end user (domain expert) from having to master any aspect of a MoC, its implementation and uses, since it is handled entirely by a language-level specification and applied systematically to any model conforming to the syntax of the language.

1.1.2 Thesis Context

This work has been conducted in the context of the *GEMOC Initiative*[7]. It is an open and international effort to develop, coordinate and disseminate research results regarding techniques, frameworks and environments to facilitate the creation, integration and coordinated use of various modeling languages used in the design of heterogeneous systems. Its goal is the *globalization of modeling languages*, that is, the use of multiple modeling languages to support the coordinated development of various aspects of a system [20].

More specifically, this thesis was funded by the *ANR INS GEMOC project*[8]. It investigates scientific issues such as the weaving of concurrency into executable metamodeling, the notions of structural and behavioral interfaces of a language, and the use of coordination patterns between languages to automatically integrate their runtimes. These research activities are concretized as a set of metalanguages to support a concurrent executable metamodeling approach, as well as a set of tool specifications for the edition and execution of models. The resulting implementation is an open-source Eclipse-based language workbench, the GEMOC Studio[9]. The partners of the ANR INS GEMOC project are: INRIA (Institut National de Recherche en Informatique et en Automatique; French Institute for

[7] http://gemoc.org/
[8] http://gemoc.org/ins/
[9] http://www.gemoc.org/studio

Figure 1.1: The ANR INS Project GEMOC logo.

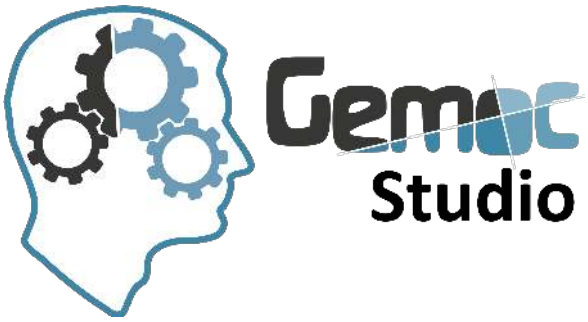

Figure 1.2: The GEMOC Studio logo.

Research in Computer Science and Automation), through IRISA[10] (Institut de Recherche en Informatique et Systèmes Aléatoires; Research Institute of Computer Science and Random Systems) and I3S[11] (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis; Computer Science, Signals and Systems Research Institute of Sophia Antipolis); Thales Research & Technology[12]; Obeo[13]; ENSTA Bretagne[14] (École Nationale Supérieure de Techniques Avancées de Bretagne; National Institute of Advanced Technologies of Brittany); and IRIT[15] (Institut de Recherche en Informatique de Toulouse; Computer Science Research Institute of Toulouse) where I conducted this thesis.

Although our work in this thesis pertains to the integration of MoCs into Language-Oriented Programming, the underlying objectives remain tied with the ANR INS GEMOC project. The possibility to integrate languages and their runtimes for the development of heterogeneous systems, and therefore interfacing languages at the structural and behavioral levels is, throughout the contributions presented in this thesis, one of the underlying purposes. Other purposes include the (graphical) animation of the execution of executable models, and the possibility to coordinate xDSMLs independently of the MoC they rely upon.

1.2 Objectives

We build upon an existing novel approach for the specification of xDSMLs [19] which makes explicit, in the execution semantics of xDSMLs, the concurrency concerns. Such

[10] http://diverse.irisa.fr/
[11] https://team.inria.fr/aoste/
[12] https://www.thalesgroup.com/en/worldwide/innovation/research-and-technology
[13] https://www.obeo.fr/en/
[14] http://www.ensta-bretagne.fr/stic/index.php/ingenierie-dirigees-par-les-modeles/
[15] https://www.irit.fr/-ACADIE-team-

xDSMLs are deemed "Concurrency-aware". These concerns are specified based on a MoC. It can be further refined for a specific execution platform (*e.g.*, to take into account any parallel facilities, or lack thereof) and analyzed for a specific model in order to assess behavioral properties of a model.

Figure 1.3 sums up the design and use of concurrency-aware xDSMLs.

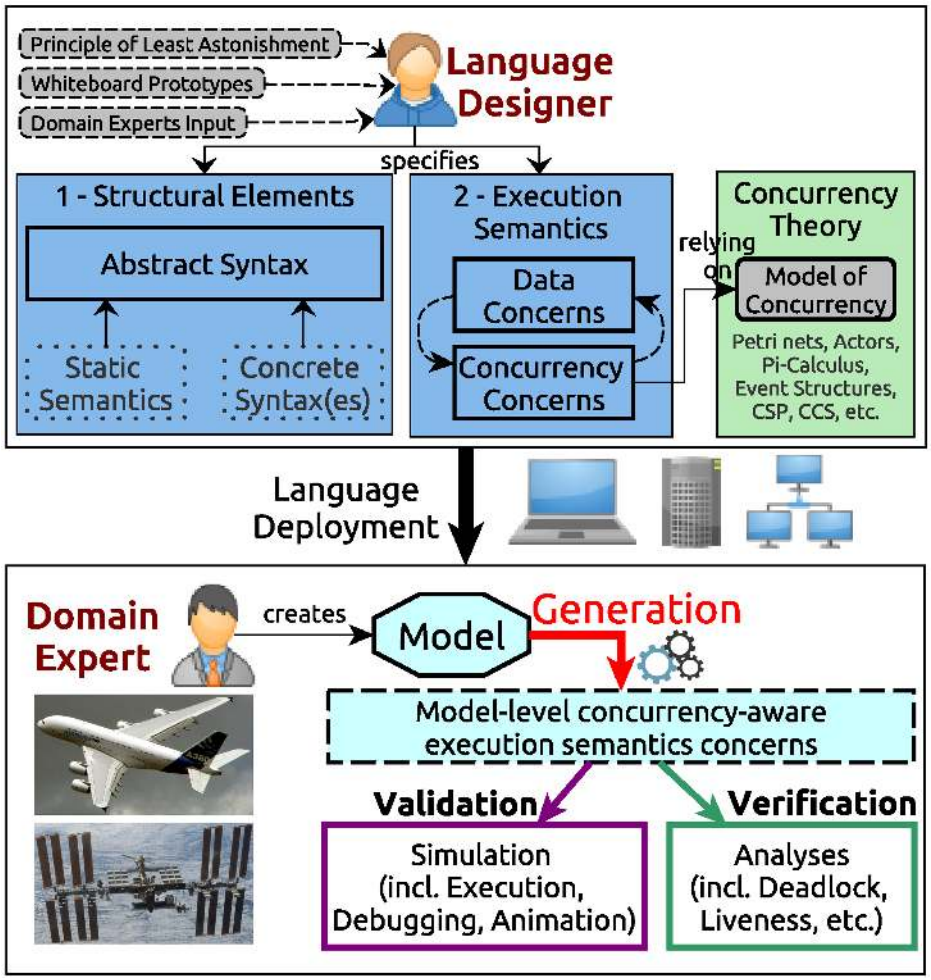

Figure 1.3: Overview of the concurrency-aware xDSML approach.

Our objectives are the following:

- To participate in the formalization of the initial concurrency-aware xDSML approach through the specification of its metalanguages.
- In particular, to identify and refine how these langages are interfaced for the definition of heterogeneous systems.

- To analyze the current limitations of the approach, in terms of what sorts of language constructs (and thus, xDSMLs) cannot be specified, or whose specification is complex to express or non-idiomatic (relative to the metalanguages used). The concurrency-aware approach should not limit the set of possible xDSMLs that can be designed with it (*i.e.*, compared to traditional language design approaches).

- To study and propose solutions to these limitations. These contributions should focus on maintaining the benefits of the initial approach (modularity of the semantics, possibility to analyze the concurrency aspects, etc.) while improving the expressive power of the approach.

- To formalize and facilitate the integration of new Models of Concurrency into the approach, in order to cater to the variety of concurrency paradigms used by different xDSMLs.

- To formalize the runtime of concurrency-aware xDSMLs, independently of the specific technologies upon which the metalanguages of the approach rely.

These objectives entail a wide range of topics. The concurrency concerns of an executable model are usually specified in an ad-hoc manner for the model. We will formalize the reification of these concerns to the language level, based on a MoC integrated into the approach. This means that the corresponding metalanguages must be defined and integrated into a language design approach. We will study how to identify and structure, in the operational semantics of xDSMLs, the concurrency concerns on the one hand; in contrast with the data aspects on the other hand. We will review the different possible interactions between these two aspects. Moreover, the initial description of the approach relies on a specific MoC called Event Structures [160]; we will discuss how to define and integrate additional MoCs into the approach, in order to cater to the different concurrency paradigms that may be required for the definition of various xDSMLs. We will also study the possibility to use translational semantics instead of operational semantics, while still making explicit the concurrency concerns.

Our work will not contribute new methods or tools to formally analyze the concurrency aspects of a system; instead, we will rely on the use of well-known formalisms developed in the Concurrency Theory community. Moreover, the executability of models is often used for *simulation* and not for production-grade *execution*. This means that the notion of *time* is not tied to the "physical" notion of time we rely on everyday. Instead, time is seen as "logical", that is, related to the notion of "execution step" during the simulation of a system (like when using the Java debugger). This may complicate the language development

activity, in the same way that debugging a multithreaded application alters its execution. The xDSMLs defined should therefore be considered for their analytical (and illustrating) purposes. Our work will not deal with the generation of an efficient implementation of the xDSMLs we specify, which is a problematic of its own.

Our contributions will be illustrated on example common xDSMLs and models, using pseudo-code and/or our metalanguage implementations to illustrate the example specifications. Our implementations of these example xDSMLs using the metalanguages developed are made available in the appendices. In particular, in our formalization of the metalanguages of the concurrency-aware xDSML approach and their implementations, we try to remain as user friendly as can be. This means that whenever possible, the metalanguages should rely on existing concepts and syntaxes of the traditional programming or modeling communities.

1.3 Outline

The rest of this thesis is structured as follows:

- Chapter 2: we give essential elements of background. We discuss the definition of concurrency and how it is specified. We also present traditional approaches to language design, as well as model-based approaches to language design. Then, we introduce early work on approaches that combine these two domains, and which constitute the initial inspiration for our work.

- Chapter 3: we present our formalization of the operational semantics approach for the specification of concurrency-aware xDSMLs. In these languages, the concurrency concerns are made explicit thanks to a dedicated specification. We illustrate the approach on an example xDSML and gradually augment the approach with features to enable the specification of advanced language constructs, or to equip concurrency-aware xDSMLs for interfacing purposes.

- Chapter 4: we give a recursive definition of the concurrency-aware approach, by enabling previously-defined concurrency-aware xDSMLs to be used as the Model of Concurrency of other concurrency-aware xDSMLs. This provides a seamless way to define and integrate new MoCs into the approach. We identify its consequences in terms of analyzability of the language and its conforming models.

- Chapter 5: we consider the definition of the execution semantics of concurrency-aware xDSMLs in a translational manner. We analyze the costs and benefits of us-

ing translational semantics instead of operational semantics for concurrency-aware xDSMLs.

- Chapter 6: we sum up our work and propose perspectives for future research activities.

Finally, the Appendices (starting from page 215) illustrate many elements of our implementation of the approach.

"A philosopher/mathematician named Bertrand Russell [...] once wrote: 'Language serves not only to express thought but to make possible thoughts which could not exist without it.' Here is the essence of mankind's creative genius: not the edifices of civilization nor the bang-flash weapons which can end it, but the words which fertilize new concepts like spermatozoa attacking an ovum."

in *Hyperion*, by Dan Simmons (1948 – current).

2 Background

Summary

We present previous and related work on the study of concurrency and language design. We start by introducing key background elements about Models of Concurrency. In particular, how their use is usually made through language constructs, libraries or frameworks, and how this may make it hard. We then present traditional language design techniques, both concerning the syntactic and semantics aspects. We then focus on Domain-Specific Languages (DSLs) and Modeling Languages (DSMLs) by discussing their purposes and their specification. Finally, we introduce the use of Model-Driven Engineering for the design of DSLs, including previous work on the specification of executable DSLs with rich and explicit concurrency semantics, which constitute the foundation upon which our contributions will be realized.

Chapter Outline

Résumé

Nous présentons dans ce chapitre des travaux existants et connexes aux thématiques abordées dans notre thèse.

Nous commençons par discuter de la définition de la concurrence, et principalement de sa relation avec le parallélisme ; ces deux notions étant souvent confondues, parfois reliées, et rarement formellement et explicitement séparées. Nous présentons ensuite la notion de modèle de concurrence (*Model of Concurrency – MoC*) qui consiste essentiellement en la définition de formalismes adaptés à l'expressions des aspects concurrents d'un système. Ces formalismes sont généralement accessibles dans les langages de programmation tels que Java, Scala, Ruby ou Python à travers des constructions de langage, des *frameworks* ou des bibliothèques.

Nous présentons ensuite les techniques traditionnelles de conception de langages. Un langage (informatique) est généralement structuré de la manière suivante. La syntaxe abstraite regroupe les concepts du langage ainsi que leurs relations. Une sémantique statique permet de définir des contraintes supplémentaires sur cette structure. La syntaxe abstraite peut être mise en correspondance avec une représentation, généralement textuelle ou visuelle, permettant aussi sa manipulation (*i.e.*, la saisie d'un programme, ou modèle, conforme au langage). Le comportement d'un langage est donné par sa sémantique d'exécution (parfois appelée sémantique comportementale ou sémantique dynamique, ou tout simplement sémantique). La sémantique d'exécution a été l'objet de nombreux travaux de recherches et de théories. Trois grandes approches de la sémantique co-existent : axiomatique, où l'on précise l'état précédent un changement (préconditions) et l'état suivant un changement (postconditions) sous forme de propriétés ; opérationnelle, où l'on spécifie comment les valeurs dynamiques évoluent durant l'exécution; et translationnelle, où l'on transforme le programme en un programme conforme à un langage dont la sémantique d'exécution est déjà définie et connue. Nous présentons ensuite les notions d'interfaces structurelle et comportementale d'un langage et leurs utilisations respectives. Nous présentons déjà quelques limitations de cette approche traditionnelle vis-à-vis de la spécification de systèmes fortement concurrents.

Nous nous attachons ensuite à une catégorie particulière de langages : ceux dédiés à un domaine particulier, appelés langages dédiés (*Domain-Specific Languages – DSLs*). Nous exposons les raisons du développement de tels langages, ainsi que la dichotomie parmi les langages dédiés entre ceux constituant une spécialisation locale d'un langage hôte généraliste (langages dédiés internes) et ceux étant des langages à part entière (langages dédiés externes). Les langages dédiés sont essentiels à la programmation orientée langages (*Language-Oriented Programming – LOP*). Cette approche repose sur l'utilisation

combinée de nombreux langages dédiés, chacun spécialisé pour un aspect particulier du système conçu. Les outils facilitant la définition de langages dédiés, appelés ateliers de langages (*Language Workbenches*), se sont développés pour soutenir cette approche.

Pour finir, nous introduisons la notion d'Ingénierie Dirigée par les Modèles (*Model-Driven Engineering – MDE*) qui place les modèles au cœur du génie logiciel. Ce paradigme est notamment propice au développement de ce que l'on apppelle les langages de modélisation (*Modeling Languages*), souvent utilisés dans l'industrie pour leur pragmatisme et practicité pour des utilisateurs non-informaticiens. Il permet aussi le développement et l'outillage de langages, et a été utilisé dans le développement de nombreux ateliers de langages.

Nous présentons enfin les premiers travaux concernant l'intégration des modèles de concurrence dans les techniques de développement de langages à base de modèles et qui ont servi de fondations pour les contributions proposées dans cette thèse.

Nous terminons ce chapitre par une présentation du contexte technique dans lequel les travaux d'implémentation liés à cette thèse ont été développés, c'est-à-dire la plateforme Eclipse et notamment son framework de métamodélisation (*Eclipse Modeling Framework – EMF*).

2.1 Concurrency and its Specifications

2.1.1 Defining Concurrency

The definition of *concurrency* is made difficult because of its domestic meaning, as illustrated by the Wiktionnary's definitions[1]:

1. The property or an instance of being concurrent; something that happens at the same time as something else.

2. (computer science) a property of systems where several processes execute at the same time.

For comparison, that same Wiktionnary's definition of *parallel*[2] is the following:

> (computing) Involving the processing of multiple tasks at the same time.

Concurrency and parallelism are however two very different concepts. In fact, the confusion between these two terms has been the subject of many interrogations[3,4,5,6,7,8] and contributions (*e.g.*, by Simon Marlow[9], author and co-developer of the Glorious Glasgow Haskell Compilation System, GHC; by Robert Harper[10], of Standard ML fame; or by Rob Pike [11], one of the designers of the Go programming language [52]). A whole subsection is also dedicated to this in Peter Van Roy's "Programming Paradigms for Dummies: What Every Programmer Should Know" [152, Subsection 4.3].

In the rest of this thesis, we will use the following definitions:

- *Parallel* is a physical concept related to the simultaneous execution of two pieces of code (*i.e.*, on two different processors).

- *Concurrency* is a logical concept related to the dependency that exists (or not) between two pieces of code.

[1] https://en.wiktionary.org/wiki/concurrency
[2] https://en.wiktionary.org/wiki/parallel
[3] http://stackoverflow.com/q/1897993/
[4] http://stackoverflow.com/q/4844637/
[5] http://stackoverflow.com/q/3086467/
[6] http://stackoverflow.com/q/3324643/
[7] http://stackoverflow.com/q/23571339/
[8] http://stackoverflow.com/q/1073098/
[9] https://ghcmutterings.wordpress.com/2009/10/06/parallelism-concurrency/
[10] https://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/
[11] http://blog.golang.org/concurrency-is-not-parallelism

As such, parallelisms are a side-effect of concurrent situations: independent pieces of code may be executed in parallel, sometimes allowing for better performance. Two pieces of code are concurrent when there is no dependency between them; they can be executed in any order (or in parallel, if the platform is able to) without changing the meaning of the program. By extension, specifying the concurrency of a program consists in specifying the dependencies between the different pieces of code constituting the program.

In particular, this means that the concurrency aspects of a program include what is commonly known as the *control flow*, such as sequences, iterations, etc. When using General-purpose Programming Languages (GPLs), parts of the concurrency aspects are already pre-determined by the language. In most GPLs, instructions are generally executed in the order they are written in (procedures, data structures, GOTOs, etc. not withstanding). In that sense, they can be said to be sequential by default. Languages may be concurrent by default, for instance when based on the Declarative Programming paradigm. Æminium [140] is an example of a permission-based, concurrent-by-default, programming language.

This unification of the control flow and concurrency concepts is also visible when considering iterations. In "Iteration Inside and Out, Part 2" [12], Bob Nystrom (part of the Dart development team) goes into the details of internal and external iterators. In doing so, he analyzes Ruby's manner of implementing iterations, which is based on the notion of *Fibers*. Fibers are a construct most often associated with concurrency rather than with control flow (*i.e.*, it is akin to lightweight/green threads, and used to realize asynchronous operations). When two objects interact with each other (*i.e.*, via method calls), it creates a dependency in the program: the caller and the callee are supposed to be in some expected state. In the case of iteration, there are two "threads" of execution: the iterator, which provides a piece of data when asked to, and the calling context, which treats that data.

Finally, this unification can also be seen in attempts to enumerate all natures of control flow constructs. This is for instance the case of the *Workflow Patterns Initiative* [133], which has devised a classification of control flow constructs in workflow systems. In this study, the authors have identified 43 patterns describing the control flow perspective of workflow systems. They give a formal description of their semantics using the Coloured Petri-Net formalism [71]. These patterns are usually handled by a language construct (or a combination of constructs) in formalisms such as BPMN [158], UML Activity Diagrams [111], BPEL [120], etc. Some of these patterns are directly related to concurrency constructs, for instance *Parallel Split* (akin to fork), *Multi-Choice* (akin to fork with guards), etc.

[12] http://journal.stuffwithstuff.com/2013/02/24/iteration-inside-and-out-part-2/

2.1.2 Models of Concurrency

Models of Concurrency (MoCs) are formalisms dedicated to the specification of concurrent systems, or to the specification of the concurrent aspects of a system. Historically, MoCs have emerged from two different communities, so their definitions and uses are not totally unified.

On the one hand, theoretical computer science has proposed to use MoCs as formalisms to represent a concurrent system in order to reason about it, or to use it as a specification. It is often formalized using mathematics, clearly defining the analyzable properties it offers. On the other hand, the programming community has developed MoCs as high-level abstractions to facilitate the definition of concurrent programs. Indeed, programming languages used for concurrent programs must offer the constructs to exploit the underlying Operating System (OS)'s capacities. Many of them stick to mimicking the OS's capacities, leaving the programmers with the difficult task of managing their threads manually, with all the traditional issues it poses : synchronizing threads and locks, ensuring that there is no deadlocks, data races, etc. These can be difficult to develop, debug, refactor and test. Advanced "programmatic" MoCs can be implemented on top of this basic layer (often as libraries or frameworks) to provide more adapted abstractions.

Theoretical MoCs [106], when implemented are usually provided as standalone languages; while programmatic MoCs are usually integrated into a language, or available through a library or framework. In any case, the implementation determines how the MoC concepts are bound to the underlying execution platform, to potentially exploit its parallel facilities. For instance, JVM-based libraries such as Scala's and Akka's actors [58, 55] or Quasar's fibers [147], are built on top of Java Threads[13]. Java Threads are bound, *by the Java Virtual Machine (JVM) implementation*, to the underlying platform. In the case of Oracle's HotSpot JVM, a one-to-one binding is made between Java Threads and kernel threads[14].

By using a MoC, we focus on the concurrency concerns of a program, abstracting away unnecessary details to ease the reasoning about its behavior. As such, there is definitely a part of subjectivity in which MoC to use for a particular system, or in how "effective" a MoC actually is. Most likely, this subjectivity will be influenced by the programmer's knowledge of and experience with the MoC, as well as the tools and formal properties available for the MoC. Still, there may be a lot of differences between two implementations of the same MoC, making it hard to compile a definite list of existing MoCs.

[13] http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

[14] http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline

When using theoretical MoCs, it can be difficult to ensure the validity of the MoC use with regards to the system considered. If used for analysis, we need to ensure that the MoC representation of the system does correspond to the intended behavior of the system; if used as as specification, we need to be able to assert the correctness of the implementation with regards to the specification. Examples of such MoCs include:

- *Petri nets* [107, 71] are a well-known computer science formalism. Their simplicity allows for a complete mathematical understanding, enabling the verification of behavioral properties, typically through model-checking. Petri nets also have many extensions, to include hierarchy, different kinds of tokens or arcs, time constraints, etc.

- *Event Structures* [160] rely on a partial ordering over a set of events. When these events represent instructions of a program, the partial ordering represents the possible schedulings of these instructions. This makes Event Structures a practical representation for concurrent programs [126].

- *Chu Spaces* [56] are an extension of Event Structures with an algebraic structure.

- *Process Algebras* such as Communicating Sequential Processes (CSP) [68], the Calculus of Communicating Systems (CCS) [101], the π-Calculus [102] or the Join Calculus [44]. Mathematical models (and their variations including time, stochastic behaviors, etc.) are designed to represent concurrent and distributed systems.

Programmatic MoCs are made available through language constructs of a host language (usually a GPL), a library, or a framework. They are mostly used for implementation purposes, to facilitate the design of highly-concurrent programs by providing high-level concepts on top of traditional threads and locks. However, their correct use is subjected to the end-user's knowledge of the theoretical model, the implementation used, and the associated good practices. Examples of such MoCs are:

- *Threading* is a MoC often encountered in GPLs because it mimics the behavior of the Operating System (OS) [51]. In that case, Threads should not be confused with the OS-level notion of thread. These conceptual threads are also called green threads, lightweight threads, coroutines or fibers. Some implementations provide advanced ways to map them to the kernel-level threads. In C and in Java, Threads are typically mapped 1:1 to kernel-level threads. They are composed of a set of instructions to execute sequentially. They must also be coordinated to ensure no concurrent modifications to the shared memory space happens. This model poses a lot of problems [89],

mainly because of the shared memory between threads (which must be controlled finely using monitors, locks, semaphores, etc.), resulting in a lot of research work proposing solutions to stir away from this model.

- *Simple Concurrent Object Oriented Programming* (SCOOP) [98] was designed for the Eiffel programming language [136] to abstract away the use of threads and locks for concurrent programs. For Eiffel, it relies on the introduction of a new keyword `separate`, used to identify classes which execute in their own thread and synchronization points of the language.

- *Software Transactional Memory* (STM) [134] can be used for controlling the access to shared memory in concurrent programs, which is often difficult to manage and is the origin of data races. This model is inspired by database transactions.

- *The Actor model* [65] advocates representing a system using a set of actors, inherently concurrent and without shared state. Erlang [4] and Scala [58, 55] are the best-known examples of languages promoting actors as their main concurrency construct.

The dichotomy between these two sorts of MoCs is not absolute, since theoretical MoCs have been a huge influence on how concurrency is implemented in programming languages. For instance, the Actor model [65] was first designed as a theoretical model, before gaining traction with Erlang's [4], and then Scala's [58] and Akka's [55] implementations. CSP has also been a major influence for Go's concurrency model [52], or for Clojure's core.async library[15].

An introduction to MoCs in a programmatic manner can be found in Paul Butcher's *Seven Concurrency Models in Seven Weeks: When Threads Unravel* [11].

Different MoCs constitute different formalisms used to capture the concurrency aspects of a system. As such, MoCs are somewhat equivalent in that they ultimately express the same thing, albeit using different rules and under different forms. Some comparisons between MoCs have been studied, for example between Chu Spaces and Event Structures, Petri nets, CCS and CSP [56, Chapter 7]; between SCOOP and CSP [10]; between Actors and Turing machines [64]. In the programming community, MoCs regarded as successful within an ecosystem are often reproduced in other communities. Examples include the Libmill library[16] which brings Go-style concurrency to C; other libraries also bring the concept of *structured concurrency* [141] to C (*i.e.*, Libdill[17]); C++ also has its implementation of

[15] http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html
[16] http://libmill.org/
[17] http://libdill.org/structured-concurrency.html

the Actor Model (*cf.* the open source C++ Actor Framework[18]). The two main paradigms used to describe MoCs are *message passing concurrency* and *shared memory concurrency*. In the former, the "computing units" of the MoC (*e.g.*, actors, processes, etc.) do not share any memory. This removes most data races issues caused by shared memory access. Instead, they communicate by sending messages to each other in order to synchronize. This is particularly helpful to represent distributed systems. Examples of message-passing based MoCs are the Actor Model [65], Process Algebras like CSP [68] and the π-Calculus [102]. In the latter, the computing units have some shared memory and the focus is instead placed on the mutual exclusion to this shared memory (*e.g.*, through locks, semaphores, monitors, etc.). Examples of such MoCs include the Threading model [89] and STM [134].

In some particular cases, some programmatic MoCs are embedded in what is known as "Asynchronous Programming". This is often concretized by language or library constructs such as Futures[19], Callbacks[20] or Promises[21]. These are often practical synthetic constructs wrapping a concurrent computation, destined to integrate seamlessly with traditional sequential code. They are implemented on top of the core concurrent constructs proposed by the language (*i.e.*, threads in Java, fibers in Ruby, etc.).

2.1.3 Shortcomings

MoCs are difficult to use because historically, they have been designed, implemented and used by different communities. Theoretical MoCs are usually provided as standalone languages, but this complicates their integration into a codebase, which usually involves specific development, integration and execution tools and particular performance objectives. Programmatic MoCs are very dependent of their embedding in a host language, making implementations of a same MoC actually difficult to compare (*e.g.*, Actors in Erlang [4] and in Scala/Akka [58, 55]). Moreover, they require a good knowledge of the theoretical model, of its implementation, and of its potential quirks (*i.e.*, depending on the host language, some concepts may be more or less verbose to express, or affect the runtime performance of the program). Additionally, there is no common interface for MoCs, so replacing one by another must always be done in an ad-hoc manner. This makes comparing MoCs difficult, because a program is always inherently highly coupled with the MoC used.

In this thesis, we will provide solutions to both of these issues for a particular class of computer languages.

[18] http://www.actor-framework.org/
[19] http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html
[20] http://www.w3schools.com/jquery/jquery_callback.asp
[21] http://clojuredocs.org/clojure.core/promise

2.2 Traditional Language Design

We present the main components traditionally constituting a computer language.

2.2.1 Abstract Syntax

The *Abstract Syntax* (AS) of a language defines the structure of valid programs. It captures the concepts of the language and the relations between the concepts (*i.e.*, the *data model* of the language), as a graph data structure. Programs conforming to the language are captured as *Abstract Syntax Trees* (ASTs), although in most formalisms they are actually graphs, and respect the structure defined by the AS.

The AS is often enhanced with what is called the *Static Semantics* of the language, sometimes to the point where "AS" designates the AS with the static semantics included. The static semantics define additional rules and constraints to the AS, restricting the set of valid programs. These rules may be difficult, or even impossible, to capture in the structure of the AS.

2.2.2 Concrete Syntax

The AS is designed with the purpose of capturing, for the computer, the structure of programs (*i.e.*, the *grammar* of the language). This comprises two responsibilities: defining the set of valid programs, but also how to store them in memory. A *Concrete Syntax* (CS) serves the same purpose for the *user*, *i.e.*, how a program can be edited, and how it is presented to the user. For instance, comments are not necessarily included in the AS because they generally are part of a sociological process which is not relevant for the computer. The CS typically defines the keywords, symbols, layouts, etc. used to edit and visualize programs.

The relation between the AS and the CS of a language is based either on *parsing* or on a *projection* [46]. The former consists in analyzing a program expressed using the CS to construct the corresponding AST. Traditional *textual* concrete syntaxes are the typical example of this approach: a program is stored as a sequence of characters, transformed into a sequence of tokens by a lexer (also known as scanner or tokenizer), and built into an AST by the parser. For historical reasons, this is the approach used by most computer languages. Nowadays, parsers can be generated based on a more abstract description of the CS. This is, for instance, the case for ANTLR (Another Tool For Language Recognition) [121] or Xtext [7]. For projection-based approaches, the AST is built directly by actions in the editor (through an API made available by the AS). The CS consists in projecting the elements of

the AST onto visual elements in the editor. These elements can be textual, similar to what is done using parsing technologies, possibly enhanced with mathematical notations like in embeddrr [156] (built on top of the JetBrains MPS Language Workbench [12]), or tailored to a certain audience (*e.g.*, young people in Scratch [129]); or graphical (*e.g.*, Simulink [93], UML [111], etc.). For instance, Eclipse Sirius [37] can be used to define such graphical concrete syntaxes.

Concrete syntaxes are usually tooled with dedicated editors, providing features designed to facilitate the user's experience with the language: syntax highlighting, refactoring, auto-completion, etc.

A language may have no CS, for instance in the case where it is only used as an intermediary format, and is never shown or modified by a user, in which case its "visual" representation is never needed. Most of the time however, languages have at least one CS, and sometimes multiple. Having multiple CSs can be used to propose different viewpoints on a same program (*e.g.*, graphical CSs can be used to get a better grasp of the structure of a program, while textual CSs are usually better to manage all the details of an algorithm; CSs can also be adapted to fit a particular user preference, such as translated keywords or different pictograms for cultural reasons, etc.). Going from the AST to its concrete syntax representation is sometimes referred to as "pretty printing".

Figure 2.1 shows an example program[22] conforming to a language modeling entities and properties, as an abstract syntax tree, and using two different concrete syntaxes: a graphical one (inspired from UML class diagrams) and a parsing-based textual one (based on curly brackets). The AST is the internal representation, by the computer, of the program; while the other two are used by the user for editing or visualizing the program.

2.2.3 Execution Semantics

The *Execution Semantics* of a language attaches a behavior to its constructs (*i.e.*, how they evolve during execution time). They are also sometimes called "dynamic semantics", "behavioral semantics", or even just "semantics". More formally, they establish a *Semantic Mapping* between the AS and a *Semantic Domain* (the concepts that exist in the universe of discourse, *e.g.*, assembly code, Java bytecode, etc.). Their specification has been the study of numerous research ever since the inception of computer science. Nowadays, we traditionally identify three main approaches to the specification of the execution semantics of a language: Axiomatic, Operational and Translational.

[22] *cf.* http://www.eclipse.org/Xtext/documentation/

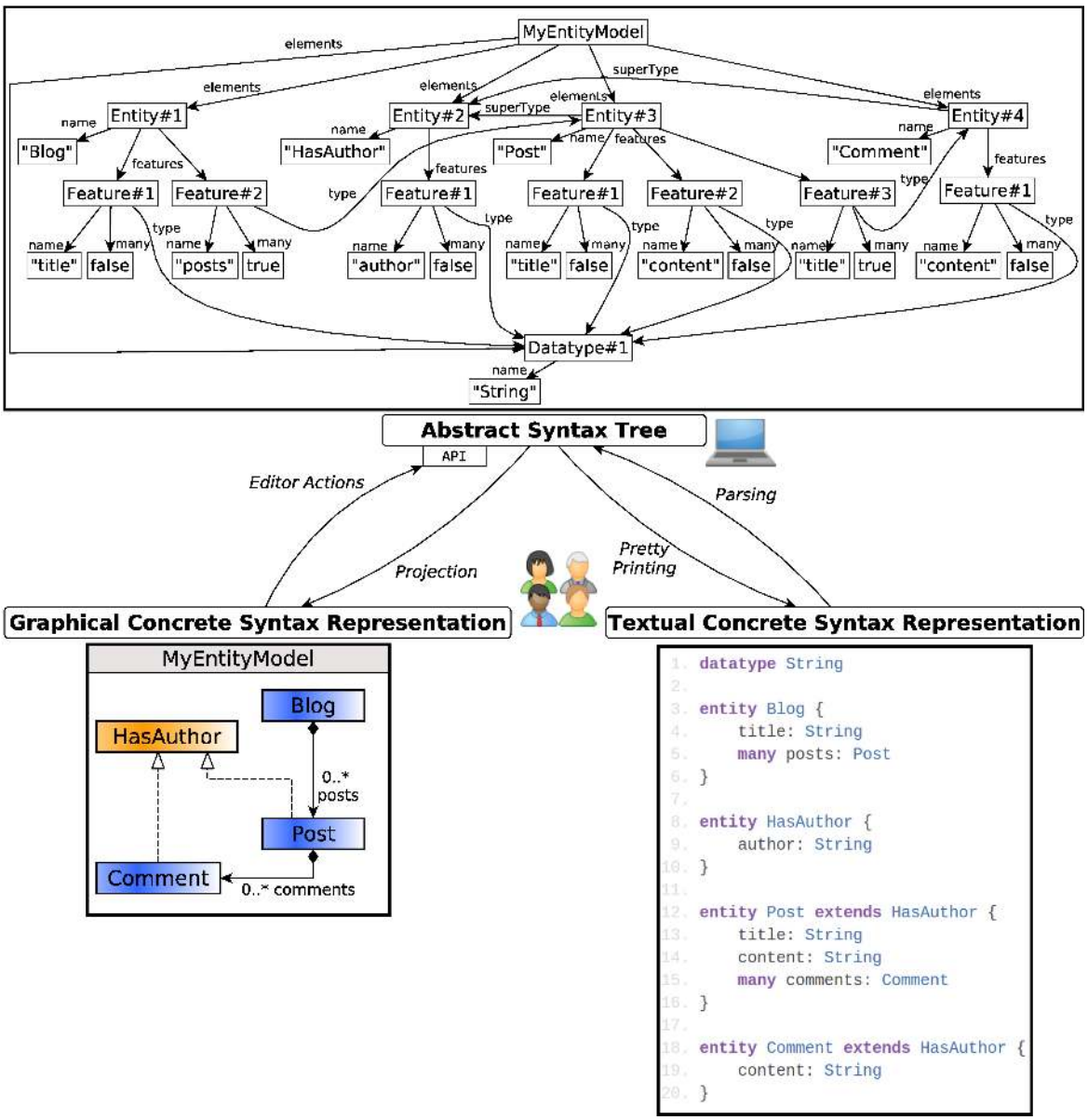

Figure 2.1: An example program represented as an abstract syntax tree (internal representation for the computer) and with a graphical and a textual concrete syntaxes (for the user).

Axiomatic Semantics

In *Axiomatic Semantics*, the meanings of a language construct are specified through properties of the program's execution state (value of a variable, current instruction, etc.) before, and after, a semantic action [31]. The best-known logic for this is the *Hoare logic* [67]. The actions are used to specify the effect, on the program's execution state, of the execution of

the language constructs. Such semantics allow reasoning rigorously about the correctness of programs and automatic generation of a correct program based on its axiomatic specification (*e.g.*, for performance or practical reasons), but do have some limitations in terms of side effects, scoping rules, etc.

Operational Semantics

Operational Semantics relies on a specification of how to perform a computation, rather than what the effects of the computation on the program state are. Operational Semantics are usually classified into two categories: *Structural Operational Semantics* [125, 103] and *Natural Semantics* [76]. In the former, each individual step of the computation is detailed. The behavior of a program is thus defined as the behavior of its parts. In the latter, only the overall computation is specified.

Translational Semantics

Finally, the execution semantics of a language can be given simply as a translation to another previously well-defined language. This technique is called *Translational Semantics*, where a source language's meanings are given entirely through the meanings of a target language. A particular case of this technique is the *Denotational Semantics* [100], when the language used is a mathematical denotation (*e.g.*, λ-calculus and the fixed point theory, etc.). In some other cases, this technique is also called *Compilation*, typically when the target language is a less abstract language such as machine code. In other cases, this is also known as *code generation*, for instance when the target language is quite high level like programming languages.

2.2.4 Semantic Variation Points

Semantic Variation Points (SVPs) are language specification parts left intentionally underspecified to allow further language adaptation to specific uses. SVPs are usually identified informally in a language's syntax and semantics specification documents. They are the acknowledgement, by the language designer, that variations can be applied to the language depending on its intended use, or to comply to specific constraints (*e.g.*, being able to run on particular execution platforms, or ensuring no undefined behaviors are allowed). SVPs can then be implemented through further refinement of the language specification or by making arbitrary choices in the implementation. For instance, in UML [111], stereotypes or profiles can be used to extend the language to fit a certain type of applications. In programming languages, such mechanisms are often implemented in an ad-hoc manner,

making difficult their study and their variation. This makes the communication between developers, and between tools, difficult.

Examples of SVPs include the following.

In the C programming language[23] specification [75], four types of SVPs are identified formally:

- Implementation-defined behavior: unspecified behavior where each implementation documents how the choice is made
- Locale-specific behavior: behavior that depends on local conventions of nationality, culture, and language that each implementation documents
- Undefined behavior: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the International Standard imposes no requirements
- Unspecified behavior: use of an unspecified value, or other behavior where the International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

When a program's behavior involves one of these SVPs, it is possible that its behavior is dependent on the specific implementation used. This complicates the communication between C developers, as well as between tools that must cooperate. Each implementation must thus carefully document and specify how these behaviors are implemented. An example of unspecified behavior of C is the order in which the arguments of a function are evaluated. If some arguments include side effects, then this can affect the overall behavior of the program.

In the Java programming language[24], threads are the main source of SVPs. The Java Virtual Machine (JVM) specification document [90] does not specify how JVM threads should be mapped to threads from the Operating System (OS). In earlier versions, JVM threads were mapped $n : 1$ to OS threads. Such threads are known as "green threads", "user threads" or "lightweight threads". They are not able to benefit from the parallel facilities of the underlying OS. Since Java 1.3, most JVM implementations, like Oracle's HotSpot, map Java threads directly to system threads [118] ($1 : 1$ mapping). This feature, with the generalization of multi-core processors, has contributed to the success of the JVM as a platform.

[23] http://www.open-std.org/jtc1/sc22/wg14/
[24] https://www.java.net

In the Python[25] programming language's standard library, concurrency can be specified using threads[26] or processes[27]. Threads execute within a process, which is in turn hosted by the OS. Different threads of a process share the same memory space, while different processes of an OS have their own memory space. Depending on the implementation of Python used, these two libraries have different semantics. The reference implementation, CPython, is subject to the Global Interpreter Lock (GIL)[28] which prevents multiple threads of the same process from running in parallel. This hinders any data races, but also prevents applications from exploiting the parallel facilities of the execution platform. In CPython, processes are thus the preferred construct for programs which seek to exploit the parallel capacities of a platform. In the Java implementation, Jython, threads are mapped to Java threads. Depending on the JVM used, the program may thus be executed in parallel.

A similar issue is found in the Ruby programming language[29]. The reference implementation (Matz's Ruby Interpreter – MRI) is subject to a GIL, while its Java implementation (JRuby[30]) can benefit from the JVM implementation's capacities of exploiting the parallel facilities of the underlying platform.

2.2.5 Language interfaces

Most computer languages are defined programmatically, *i.e.*, they are "programs" themselves (defined using metalanguages), and live within a technological ecosystem, generally equipped with other computer languages. As such, they can interact with, or be the subject of interactions from, other programs. Programs communicate through interfaces. For a computer language, we distinguish two natures of interfaces: structural interfaces and behavioral interfaces.

Structural Interfaces

The *structural interface* of a language deals with the syntactic aspects of the language, *i.e.*, it exposes the constituents of a program. This can be used to perform static analysis on a program's content (*e.g.*, to find duplicate or dead code, or for the type system).

[25] https://www.python.org/
[26] https://docs.python.org/3.5/library/threading.html
[27] https://docs.python.org/3.5/library/multiprocessing.html
[28] https://docs.python.org/3.5/glossary.html#term-global-interpreter-lock
[29] https://www.ruby-lang.org
[30] http://jruby.org/

Behavioral Interfaces

The *behavioral interface* of a language enables any external program to interact with programs conforming to this language during their execution. This can be exploited for several purposes, such as injecting additional code, coordinating other components, or debugging. Such interfaces are often devised in an ad-hoc manner in a language implementation, making their use tied to a particular implementation of the language. For instance, in Java, debugging informations for a class are available (if compiled with the corresponding option) at runtime, and can be exploited by IDEs to present sophisticated debug views to the user.

2.2.6 Shortcomings

Language design is well-known by now, however, the specification of languages with a focus on concurrent programs remains difficult. In the traditional approaches we have described, the concurrency aspects are either inherited from the execution platform, or from the metalanguage(s) used to specify the execution semantics; or meddled with the rest of the semantics. This makes them difficult to study, analyze and refine. Moreover, it requires a form of expertize in language design in order to be able to understand the concurrency aspects of a language. Additionally, traditional language design techniques do not handle well the specification, implementation and management of SVPs. They are often specified informally in the language specification document; implemented and documented by the implementors (if ever). Comparing them to ensure the correctness of a program independently of the implementation used is difficult.

2.3 Domain-Specific Languages

In this thesis, we focus on a particular class of computer languages: *Domain-Specific Languages* (DSLs) [47, 49].

2.3.1 Purposes

For historical reasons, General-purpose Programming Languages (GPLs) such as C, Python or Java, constitute the most popular category of computer languages. These languages are designed to be generic, and their fit for a particular problem include criterias such as the affinity of the user with that language's syntax, semantics and ecosystem; the available ecosystem of libraries, frameworks and guides that could help express the problem's solution; the correct integration of the language's runtime with existing infrastructures.

However, the complexity of modern softwares and systems tends to overwhelm the generic facilities of GPLs. It is not that they are not capable of expressing solutions for complex problems; but rather that they tend to do so in a verbose or tortuous manner, ultimately rendering complex their specification, implementation, debugging, testing, and evolution. To alleviate this issue, DSLs have been gaining traction. They aim at providing the right constructs to address problems of a specific domain. They sacrifice the genericity of GPLs in order to offer adequate syntax and semantics for a particular domain. As a consequence, the tools accompanying the language are also domain-specific, and can be made more efficient (*e.g.*, more intuitive, with domain-specific features, etc.) for the domain at hand.

2.3.2 Tradeoffs

DSLs are usually "smaller" languages than GPLs, in the sense that they focus on a single domain, may be internal to a company or to a specific set of practitioners, and therefore with a smaller userbase. Many do not even need to be Turing-complete. Their smaller size and need to evolve alongside the domain they address means that DSLs typically evolve faster than GPLs, requiring additional toolings allowing quick iterations. Table 2.1 sums up the main differences between DSLs and GPLs, in the general case.

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Table 2.1: Main differences between DSLs and GPLs. Courtesy of M.Völter [155].

GPLs take most of their characteristics from the second column, while DSLs tend to pick from the third column. It is important to not consider this table as absolutes: "Domain-specificity is not black-and-white, but instead gradual: a language is more or less domain-specific." [155]. As such, the table above should not be considered literally, but rather as a summary of the potential differences between DSLs and GPLs.

For instance, variations of the SQL language have been proven to be Turing-complete [43]. That does not mean that implementing complex softwares with it is a good idea. In the same manner, HTML, which may be seen as a Domain-Specific Markup Language, has a large, anonymous and widespread community. Python's infamous backward-incompatible changes (*i.e.*, between versions 2 and 3) is also uncharacteristic of GPLs, which usually evolve conservatively in order to cater to enterprise-grade softwares.

The domain-specificity DSLs provide must always be considered with regards to the genericity they sacrifice for it. Moreover, the additional costs of designing, developing and maintaining a DSL mean that they are not necessarily the best investment for lower-scale organizations or small problems. But "adopting an existing DSL is much less expensive and requires much less expertise than developing a new one. Finding out about available DSLs may be hard, since DSL information is scattered widely and often buried in obscure documents. Adopting DSLs that are not well publicized might be considered too risky, anyway." [96]. However, empirical studies have shown that DSLs are a more effective tool for solving problems of the domain they have been designed for [82, 80, 123, 81].

2.3.3 Internal and External DSLs

DSLs are usually designed either as standalone languages, or as GPLs extended with domain-specific concepts. The former are called *External DSLs* while the latter are called *Internal DSLs* (or embedded DSLs).

Internal DSLs are embedded into a host GPL, extending or redefining the syntax or core language constructs such that they are more adapted for a particular domain. The frontier between internal DSLs and Application Programming Interfaces (APIs) is blurry. Fluent APIs, which focus on the readability of the client code using them, can be considered as this frontier. Internal DSLs are often made possible thanks to features such as dynamic typing or operator overloading. Scala [31] and Ruby are the most glaring examples of modern GPLs used to host internal DSLs, due to the meta-facilities they provide, with Lisp[32] being their forefather. Internal DSLs are practical when they need to be integrated with an existing code base that works well with the host GPL. They can however be difficult

[31] http://www.scala-lang.org/
[32] https://common-lisp.net/

to customize or restrict for the purposes of the DSLs. For instance, DSLs are sometimes designed such that only valid programs may be entered. This is often challenged by the powerful expressive power of the host GPL. Internal DSLs may be embedded *shallowly* (*i.e.*, the language constructs are directly defined in terms of the host language) or *deeply* (*i.e.*, the language constructs are used to construct an AST, which may in turn be optimized, compiled to another language, etc.) [143].

External DSLs are full-fledged languages, which, as of today, are usually more complex and expensive to develop than internal DSLs. Internal DSLs rely on an existing syntax and semantics, only specializing or extending specific parts of the host language. Meanwhile, external DSLs need to consider traditional language design elements such as its abstract and concrete syntaxes, and the corresponding tool support. Since external DSLs are standalone language, they can be more easily customized and adequately tooled, for instance to support Integrated Development Environment (IDE) features such as syntax highlighting and refactoring, static verifications, or domain-specific features. DSLs are usually smaller than GPLs, thus their tools will also generally be simpler to produce. Still, the main issue remains in evaluating whether or not this customizability outweighs the cost and effort of designing and implementing an external DSL. In modern techniques, part of the tooling can be derived from the language definition, thus contributing to the popularization of external DSLs.

Some hybrid approaches have also been proposed, in order to facilitate the design of external DSLs which can easily be integrated with existing DSLs and GPLs. This is for instance the case of Xbase [38], which provides a base expression language, with a parser, linker, compiler, interpreter and IDE features. It can be extended via language inheritance to define new JVM languages, totally compatible with existing JVM languages such as Java or other Xbase-based DSLs.

2.3.4 Towards Language-Oriented Programming

Complex systems entail a wide range of issues, and thus often require a combination of different computer languages [149]. For instance, web development frameworks usually integrate front-end technologies (CSS, HTML and the *de-facto* standard Javascript, including sophisticated libraries) and back-end technologies (a database, queried using an appropriate query language such as SQL, and the application server implemented using a GPL such as Java, Python or Ruby). In such frameworks, there is a limited and known set of languages that must cooperate together. The GPL used for the back-end serves mainly as the glue to tie the database to the front-end.

More generally, modern softwares and systems usually involve an unknown number of different langages. Ideally, all these langages are DSLs used for each separate aspect of the system (*i.e.*, instead of one big GPL program separated into modules or packages). For particular cases, the integration of these different languages may be done in an *ad-hoc* manner (like web frameworks do); but this is difficult to generalize. Considering the fast-paced evolution of DSLs and the multitude of different concerns involved, manual integration of languages is not a sustainable solution. This challenge remains to be addressed and is identified as the problem of the *globalization* of languages [20] (*cf.* the GEMOC Initiative[33]).

Language-Oriented Programming (LOP) [157, 33] is an approach that places the use of multiple languages, most commonly multiple DSLs, at the heart of the engineering activities. By placing the focus on the multiplicity of languages, LOP incidentally raises the issue of specifying, implementing and tooling these languages [16]. Such meta-tools are called *Language Workbenches* [45, 39]. They usually embed metalanguages allowing the specification of the syntactic and semantic aspects of languages. For the former, language workbenches can provide additional assistance in terms of IDE integration, *i.e.*, automated syntax highlighting and editor features can be inferred automatically from the syntaxes. The semantic aspects can be specified in different manners (axiomatic, operational, translational, etc.) and interpreted or compiled.

Language Workbenches are not new. Early iterations of language workbenches include MetaPlex [13], CENTAUR [8], Metaview [137], MetaEdit [135], the Cornell Program Synthesizer [128], or ASF+SDF [79, 150]. But with the technological evolutions of language design techniques and IDE platforms, they can now integrate powerful IDE features without significant effort. A comparison of modern language workbenches can be found in the different editions of the *Language Workbenches Contest* [84]. Examples of modern language workbenches include Jetbrains MPS [12], Spoofax [77], MetaEdit+ [146], the Diagram Predicate Framework (DPF) Workbench [83], the Rascal Language Workbench [151], or Microsoft's Modeling SDK for Visual Studio (MSDK) [21].

2.3.5 Shortcomings

In LOP, the multiplicity of DSLs employed is tackled by the language workbenches which provide the tools and methodologies to define DSLs using appropriate metalanguages, and help with their tooling by generating part of their IDE integration, static verification, etc. They also often come with the means to specify the execution semantics of the DSLs. How-

[33] http://www.gemoc.org/

ever, like traditional language design techniques, they do not focus on the concurrency aspects of the execution semantics, thus making complex the specification and analysis of DSLs for highly-concurrent systems. We can draw inspiration from another discipline, *Multi-Paradigm Modeling* (MPM) [104, 50, 59], which tackles the use of several formalisms to specify heterogeneous systems. The formalisms used usually rely on different concurrency models due to their heterogeneous nature (*e.g.*, signal processing, electronics, hydraulics, etc.). But MPM tools and approaches, such as Ptolemy [127], ModHel'X [60, 9], AToM^3 [26]; and approaches based on Discrete Event System Specification (DEVS) [48] often embed and rely on well-known existing formalisms, and defining and integrating new ones is a complex task. In this thesis, we will work on providing a language workbench adequate for LOP, while making explicit the rich concurrency features of the execution semantics of the DSLs, based on MoCs that can be defined and integrated seamlessly into the language workbench.

2.4 Model-Driven Engineering for Domain-Specific Modeling Languages

2.4.1 Model-Based Software Engineering

To palliate the growing complexity of systems, (software) engineering approaches have evolved to include the use of *models*, leading to what is called *Model-Based Software Engineering* (MBSE). In this approach, models are used to represent an aspect of a system, abstracting away unnecessary details, to help reason about it. Models conform to a *metamodel*, that is, a model describing the structure of models.

Models may be used in several manners. They can serve as a communication and documentation artefact, as a mere blueprint or specification, or used to drive the engineering process (for instance through code generators). In the latter case, we call this approach *Model-Driven Engineering* (MDE). MDE entails all of the traditional engineering activities: designing, programming, testing, validating, etc.

The Object Management Group (OMG)[34], which standardizes object-oriented and modeling technologies, has formalized its approach of MDE in what is called *Model-Driven Architecture* (MDA) [117]. Individual standards may also be used independently, most notably:

[34] http://www.omg.org/

- MOF/EMOF: (Essential) Meta-Object Facility [112]. MOF is the OMG's meta-metamodel, that is, a metamodel used to define metamodels. MOF is *metacircular*: MOF can be defined using MOF.
- XMI: XML Metadata Interchange [115]. XMI is the OMG's XML-based format used to store models whose metamodel conforms to MOF.
- OCL: Object Constraint Language [113]. OCL is the OMG's declarative language designed to express constraints and object query expressions on MOF models and metamodels.
- QVT: Query/View/Transformation [114]. QVT is the OMG's set of standard languages for model transformations.
- MOFM2T: MOF Model To Text Transformation Language [109]. MOFM2T is the OMG's standard language for transforming models into text.

MBSE and MDE still have many challenges to overcome before becoming the general paradigm for software engineering. For instance, in the spaceflight software domain [124], these challenges include: a lack of coordinated development approach, making difficult the comparison between MBSE tools and methodologies, or the consistent adoption by a group of practitioners; the integration of multiple model-based languages, like for LOP; the conformance of the model to the real-world system (*e.g.*, for verification and validation purposes); the consistency between the model and the generated code (*i.e.*, certifying code generators is technically, and sociologically, difficult); etc. Still, they have become a popular paradigm for some engineering fields such as systems and controls engineering (Simulink [93], SCADE/Lustre [57], Arcadia/Capella[35] [132]) or database systems [145].

2.4.2 Modeling Languages

MBSE and its specializations rely on the use of models, and of metamodels to describe the structure of models. The similarities between, on one hand, metamodels and abstract syntaxes, and on the other hand, models and programs, have lead to the use of MDE technologies for the development of *Modeling Languages* (MLs). When dedicated to a certain application domain, these languages are thus said to be *Domain-Specific Modeling Languages* (DSMLs). Other MLs are usually said to be General-purpose Modeling Languages (GMLs), such as the Unified Modeling Language (UML) [111]. Actually, GMLs are often constituted of several different modeling languages, each with a focus on a certain aspect

[35] https://www.polarsys.org/capella/

or with a particular view of the system. This is the case of UML, made up of Structure Diagrams (Class Diagram, Object Diagram, Package Diagram, Component Diagram, etc.) and Behavior Diagrams (Activity Diagram, State Machine Diagram, Sequence Diagram, etc.), or of Simulink [93], whose main diagrams are block-based dataflows (with blocks issued by various libraries, often dedicated to a particular domain like physics modeling, control systems, communications, real-time systems, etc.), but which also supports state machines or discrete-event simulations. In that sense, most GMLs can be considered as a set of interoperable DSMLs.

MLs rely on powerful abstractions to represent in a manner relevant to a particular purpose, a system. DSLs provide constructs facilitating the expression of solutions of a particular domain. DSMLs are thus both adequate to solve problems of the domain they were designed for, while abstracting away unnecessary details of the system. An important consequence is that the *usability* of such languages should be optimal: the language constructs make it easy to specify solutions, and are meaningful for domain experts. DSMLs have proven effective at solving problems of the domain they have been designed for [78]. By construction, this makes them the "best tool for the job". DSMLs can also be considered as an implementation of what is called *Domain-Driven Design* [41], which advocates placing the core domain and its logic at the center of the software development activity, based on a collaboration between technical experts (in the context of LOP, language designers) and domain experts.

In a Model-Driven approach, a language's AS is captured as a metamodel; programs are captured using XMI; and static semantics are specified using OCL.

2.4.3 Executability

When adjoined with an execution semantics, DSMLs are said to be *eXecutable* (xDSMLs). Like for traditional computer languages, the execution semantics can be specified using several techniques, denominated in this context as *Executable Metamodeling* [105, 15] techniques. They are usually inspired from the main semantics approaches we have described: axiomatic, operational and translational. Examples of such approaches include the Executable DSML pattern [17], xMOF [94], Maude [131], or Kermeta [72].

Models and metamodels are often rooted in a GPL (for historical reasons or for developing associated tools such as IDEs or code generators), therefore the metalanguages used to specify the execution semantics are often based on that GPL too. For instance, UML has historically been developed in a Java/JVM environment. Its Activity Diagrams can be executed according to the foundation Subset for Executable UML Models (fUML) [116],

whose semantics is given in English and in Java (as a reference implementation[36]). The execution semantics of xDSMLs may also be defined in a translational manner, for example through an implementation of the OMG's QVT (*e.g.*, the ATLAS Transformation Language (ATL) [74, 73]) to define the translation from an xDSML to another xDSML with execution semantics already defined.

2.4.4 The Inception of Concurrency-aware xDSMLs

In "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software" [142], Herb Sutter, of C++ fame, describes how CPU designers are confronted with the limits of physics (notably, in terms of heat production and energy consumption) and its impact on software engineering. In particular, the computer languages used for writing softwares are concerned: they must provide sophisticated tools for adequately expressing the concurrency aspects of complex softwares and systems, and enable the use of the parallel facilities of the execution platform they are deployed onto.

In this thesis, we propose to bridge the chasm between Language-Oriented Programming, *i.e.*, the design of xDSMLs in a language workbench, and the paradigm shift resulting of the end of the "free lunch", *i.e.*, the integration of Models of Concurrency into their execution semantics. This is synthesized in the design of so-called *Concurrency-aware xDSMLs*. Herb Sutter published an update to his "free lunch" article[37] in which he identifies that "Programming languages and systems will increasingly be forced to deal with heterogeneous distributed parallelism". By making their use of a MoC explicit, concurrency-aware xDSMLs can be designed agnostic of any execution platform's parallel capacities, and refined only at the deployment phase. This characteristic is made possible by the domain-specificity of the language. The explicit use of a MoC at the language level is structured in the *separation of concerns* advocated by the concurrency-aware xDSML approach. In this separation of concerns, the data and operational aspects of the execution semantics are separated from the concurrency aspects which are captured based on a particular MoC.

First results towards this goal were published by Benoit Combemale *et al.* in the International Conference on Software Language Engineering 2012 [18] and 2013 [19].

In [18], the authors present an approach to reconcile Metamodels, used to capture domain-specific concepts and their actions, with "Models of Computations", used to orchestrate the actions of a domain-specific model. Both concepts have been developed in independent research communities: the former in the Model-Based Software Engineering and Domain-Specific Languages Design communities; the latter in the Concurrency The-

[36] https://github.com/ModelDriven/fUML-Reference-Implementation
[37] https://herbsutter.com/welcome-to-the-jungle/

ory community. The main difficulty consists in identifying, in the execution semantics of DSLs, which parts belong to the domain-specific actions, and which parts belong to the Model of Computation. The latter are captured using ModHel'X [60], a framework for building and executing multi-paradigm models. It uses a generic abstract syntax to capture the models, but the execution semantics is based on rules defining the semantics of control and concurrency between the elements of a model. Figure 2.2 shows the proposed separation of concerns of the semantic mapping between the AS and the Semantic Domain of a DSL.

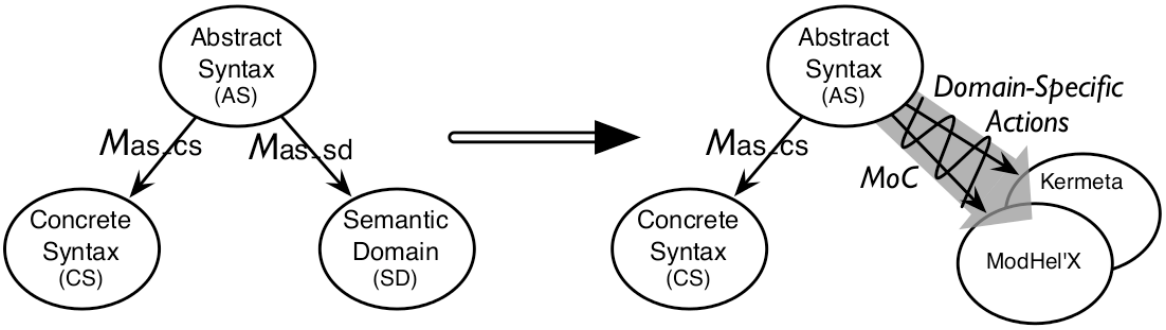

Figure 2.2: Separation of concerns in the execution semantics of DSL proposed in [18].

In [19], the authors improve the previous approach by identifying the need for an explicit coordination of the language concerns identified previously. The concurrency concerns are captured thanks to a specification of events with causal and temporal relationships between them, inspired from Event Structures [160]. These abstract events (from the concurrency concerns) are then mapped with concrete actions (in the Domain-Specific Actions – DSA) by a coordination specification called the Domain-Specific Events (DSE). At runtime, it enables using the execution of the event structure to coordinate the domain-specific actions resulting in changes in the model. They also describe the architecture of the language workbench and of the generic execution engine for concurrency-aware xDSMLs. Figure 2.3 shows the xDSML design approach proposed.

The main contribution of the concurrency-aware xDSML approach proposed in [18, 19] consists in the separation of concerns of the execution semantics of xDSMLs. In particular, the explicit identification of the concurrency concerns, using an appropriate and dedicated formalism based on a Model of Concurrency, enables its refinement, variation, and analysis. Refinements can be exploited during the deployment of the language to a specific platform, in order to specialize the language to the platform. Variations can be used to adapt the language to different communities, purposes or uses. Analyses can be performed on the model-level specifications to assess behavioral properties of the systems being mod-

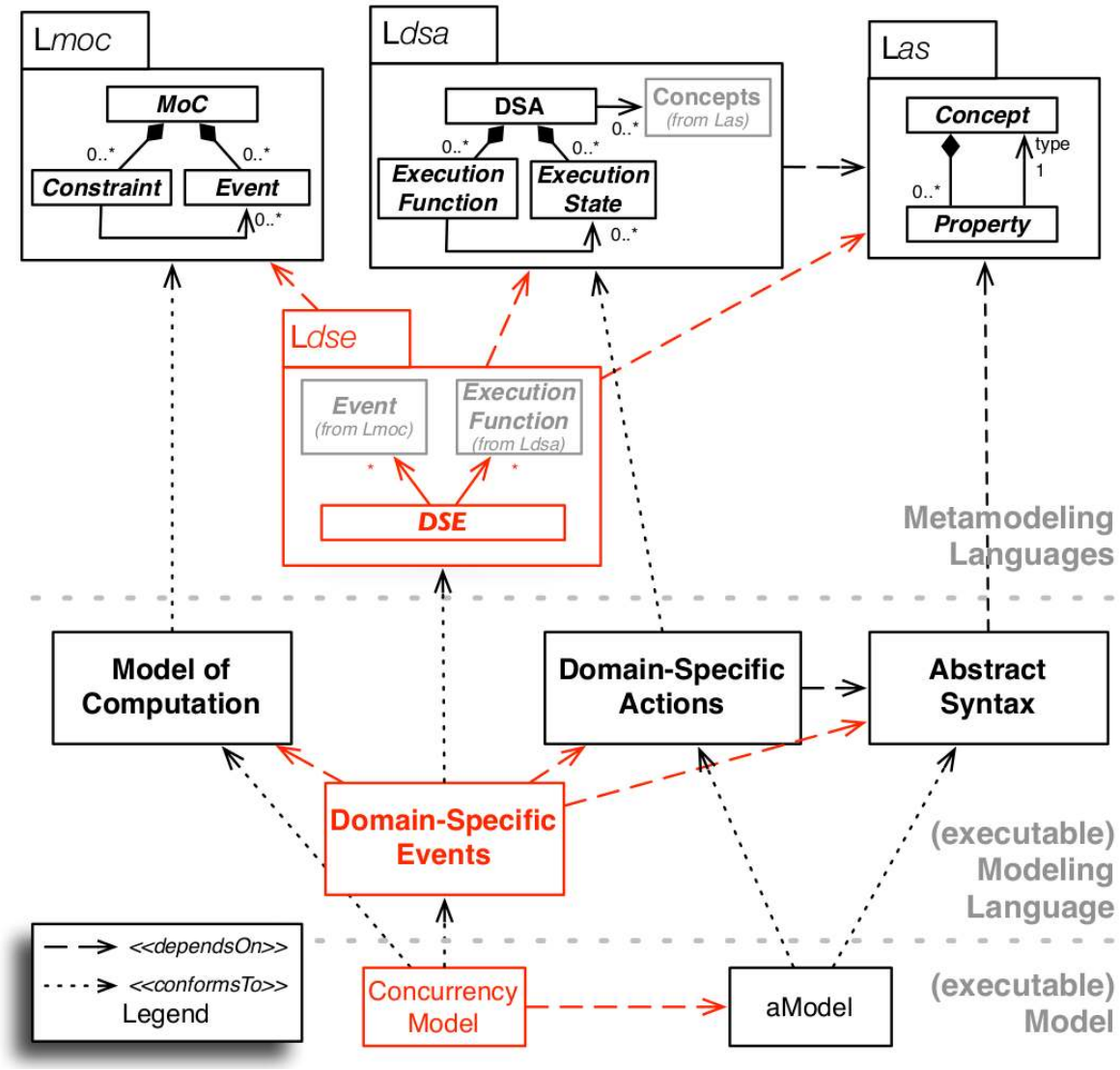

Figure 2.3: Modular design of concurrency-aware xDSMLs as proposed in [19].

eled. From a concurrency theory point of view, the approach enables the systematic use of MoCs at the language level, whereas MoCs usually have to be used through language, framework or library constructs, which usually requires particular training or knowledge about an implementation. In the concurrency-aware approach, this use is defined at the language level, therefore removing from the domain expert the responsibility to select or use a MoC.

In the rest of this thesis, we will build upon the description of the approach from [18] and [19] to formalize, improve and extend the design and execution of concurrency-aware xDSMLs. In particular, we address some existing problems of [19]:

- **Multiplicities of the relations between concerns**: the multiplicity of the association between domain-specific events and actions is left unspecified. It is not clear what is the exact semantics of several domain-specific events mapped to one domain-specific action; or how one domain-specific event mapped to multiple domain-specific actions should behave.

- **Restriction of the concurrency concerns**: it it said that the partial ordering can be restrained due to the call to some execution function, however it is not clear how this restriction is specified, and how it is realized at runtime.

- **Model of Computation or Concurrency**: in [18] and [19], the term used for the concurrency concerns is "Model of Computation". In the literature, the relation between "Model of Computation" and "Model of Concurrency" are not clear: they are often used interchangeably (*e.g.*, "Model of Concurrency or Computation (MoC)" in [54]). The π-Calculus [102] is said to be a "model of computation for concurrent systems" [159]. Traditionally, Models of Computation were developed in the computation theory field, in a time where parallel architectures were not mainstream. Considering the definition of "concurrency" used in this thesis, as presented in Section 2.1, sequentiality is a special case of concurrency, explaining why Models of Computation such as the λ-calculus [14, 6] can be encoded in theoretical Models of Concurrency like the π-Calculus [159]. In the rest of this thesis, we will only use the term "Model of Concurrency" because we focus on specifying the concurrency concerns of an xDSML, explicitly separated from its data concerns.

2.5 Technical Context

The technical efforts presented in this thesis have been implemented in an Eclipse-based language workbench developed for the ANR INS Project GEMOC, called the GEMOC Studio. We introduce the main technologies used to build this language workbench.

2.5.1 The Eclipse Platform

The Eclipse Platform is an open-source platform, originally designed for the development of IDE products, although it has evolved onto a framework for developing general-purpose applications through its Rich Client Platform (RCP). It is overseen by the Eclipse Foundation[38]. At its core, Eclipse is constituted of a small runtime kernel, and most of its features

[38]https://www.eclipse.org/org/foundation/

are implemented as Eclipse plugins. Eclipse's Equinox is the reference implementation of the Open Services Gateway initiative (OSGi), a standard that implements a component model platform for the Java/JVM environment.

Thanks to this modular architecture, Eclipse can easily be extended with additional features. In particular, many plugins have been developed to implement IDEs for computer languages such as Java, C, Python, Ruby, PHP, Prolog, Scala, etc. It also supports different version control systems such as SVN[39], Git[40] or Mercurial[41].

2.5.2 The Eclipse Modeling Framework

One particular contribution of Eclipse is its Modeling Project [53], which includes a wide range of features related to modeling technologies. At its heart is the Eclipse Modeling Framework (EMF) [36]. The core EMF and EMF-based technologies relevant to our work are the following:

- Ecore [34], the *de facto* reference implementation of the OMG's EMOF [112]. Figure 2.4 shows the hierarchy existing between the main Ecore components.
- Eclipse OCL [35], an implementation of the OMG's OCL [113], enabling the definition of static semantics for Ecore metamodels.
- Sirius [37], an editor to create graphical modeling editors for Ecore metamodels.
- Xtext [7], a framework which eases the definition of textual concrete syntaxes for Ecore metamodels. Includes the automatic generation of an ANTLR specification (to generate a parser) and of IDE features within Eclipse.
- Xtend [7], a JVM-based GPL which compiles to readable Java code. Its syntax is consistent with Java's for ease-of-adoption, while adding a lot of features to make it less verbose (*e.g.*, var/val keywords, lambdas, advanced collection operations, etc.). Its *Active Annotations* feature allows developers to easily inject additional code automatically during the compilation phase.

Our work was implemented on top of the GEMOC Studio, which includes these technologies as well as other EMF-based technologies built by the project's various partners. For each chapter, the relevant ones are detailed in their "implementation" subsection.

[39] https://subversion.apache.org/
[40] https://git-scm.com/
[41] https://www.mercurial-scm.org/

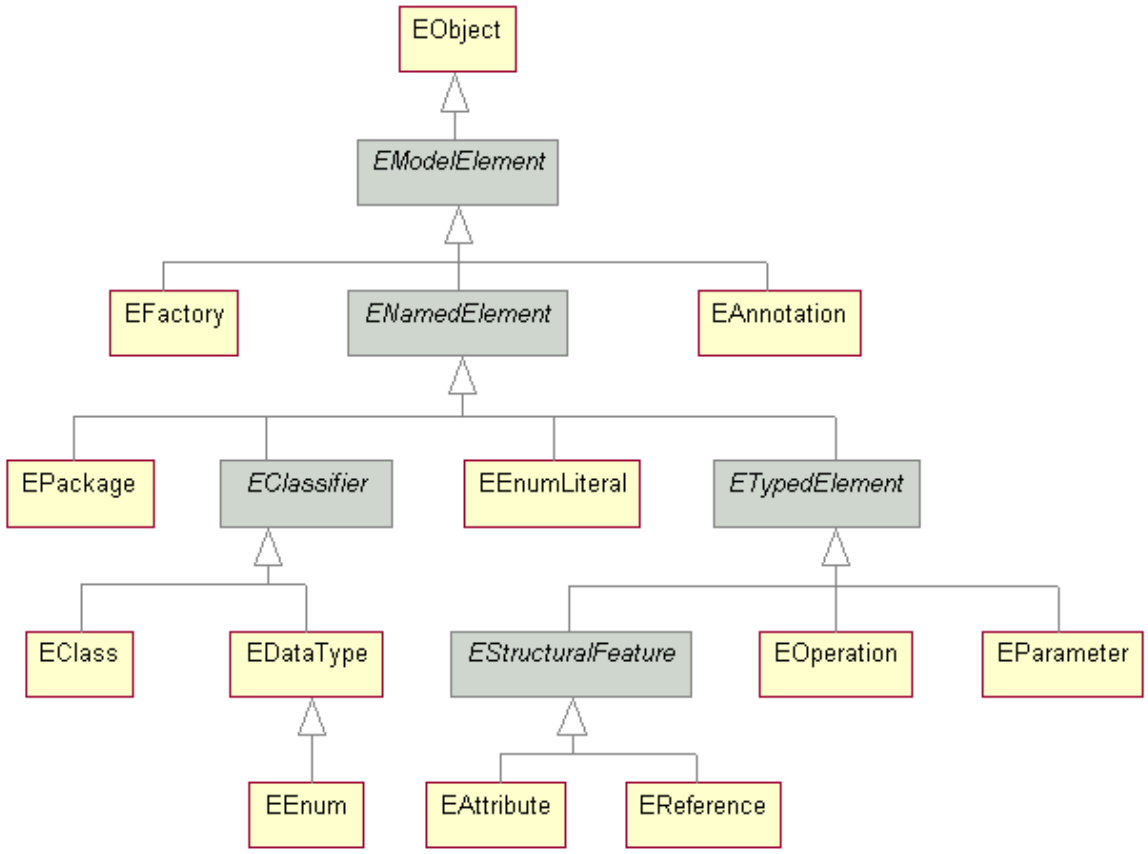

Figure 2.4: Hierarchy of the main Ecore components.

"It is at this point that normal language gives up, and goes and has a drink."

in *The Color of Magic*, by Terry Pratchett (1948 – 2015).

3

Design of Concurrency-aware xDSMLs

Summary

We refine the existing concurrent executable metamodeling approach enabling the definition of so-called concurrency-aware xDSMLs. We start by illustrating shortcomings of the approach on an example xDSML, fUML. We then refine the approach by formalizing it, in particular the separation of concerns upon which it is built. We detail the responsibility of each concern, how they are specified and how their respective runtimes work. Based on these foundations, we then refine the shortcomings of the approach and propose features to complete the approach. For each issue, we identify the associated challenges and present the requirements as constraints for the solution. Finally, we give the architecture of our implementation of the approach in an Eclipse-based language workbench.

Parts of the contributions presented in this chapter have been published in the *8th ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2015) [85] and in the *1st International Workshop on Executable Modeling* (EXE 2015) [86].

Chapter Outline

Résumé

Ce chapitre présente le cœur de notre travail. Nous formalisons et étendons une approche de métamodélisation exécutable et concurrente, permettant la création de langages de modélisation dédiés exécutables avec utilisation explicite et systématique d'un modèle de concurrence (*Concurrency-aware eXecutable Domain-Specific Modeling Languages*).

Nous nous intéressons à la spécification de la sémantique opérationnelle de ces langages. De fait, les problématiques liées à la spécification de la syntaxe abstraite, des syntaxes concrètes et de la sémantique statique sont considérées comme ayant été résolues en amont, selon le principe de séparation des préoccupations. Le langage *Foundational Subset for Executable UML Models (fUML)* [116] nous servira à illustrer l'approche. En particulier, dans fUML, la spécification de la sémantique ne détaille pas comment exécuter les branches concurrentes (c'est-à-dire comprises entre un *ForkNode* et un *JoinNode*). Ces branches peuvent donc être exécutées en parallèle, en séquence, ou selon tout autre arrangement. Ce choix est en général implicite car inscrit directement dans l'implémentation, peu documenté et difficile à modifier. Les *concurrency-aware xDSMLs* rendent explicites ces choix à l'aide d'un formalisme adapté, facilitant leur spécification, leur analyse, ainsi que la spécification, l'implémentation et la gestion de différents points de variation sémantique (*Semantic Variation Points – SVP*).

L'approche que nous formalisons repose sur une séparation des préoccupations au sein de la sémantique opérationnelle. Celle-ci est donc séparée en trois parties : les règles sémantiques (*Semantic Rules*), l'utilisation d'un Modèle de Concurrence (*Model of Concurrency Mapping – MoCMapping*), et un protocole de communication (*Communication Protocol*) connectant les deux premières parties. Les *Semantic Rules* (correspondant aux *Domain-Specific Actions* proposées dans [18, 19]) étendent la syntaxe abstraite du langage avec les données d'exécution (*Execution Data*), représentant l'état courant du modèle durant son exécution, et les fonctions d'exécution (*Execution Functions*), définissant comment les *Execution Data* évoluent durant l'exécution. Par exemple dans fUML, les arêtes entre les noeuds portent des jetons (*Tokens*), et ces jetons sont consommés, transférés, dupliqués ou produits par l'exécution des noeuds (en fonction de leur nature concrète). Le *MoCMapping* définit l'utilisation systématique d'un Modèle de Concurrence (*Model of Concurrency – MoC*) par tout modèle conforme à la syntaxe abstraite du *xDSML*. La concurrence de tout modèle conforme à la syntaxe abstraite du langage sera ainsi représentée sous forme de modèle conforme au *MoC* utilisé. Cette spécification est appelée l'application du modèle de concurrence (*Model of Concurrency Application – MoCApplication*). Le *MoC* utilisé initialement dans l'approche repose sur les structures d'évènements (*Event Structures*) [160]. Le formalisme utilisé pour spécifier le *MoCMapping* est en conséquence appelée structure

de types d'évènements (*EventType Structures*). Une *Event Structure* définit un ordre partiel sur des évènements qui représentent des actions abstraites. Cette représentation de la concurrence, indépendante de l'état courant du modèle, la rend analysable par des outils dédiés pour la vérification de propriétés comportementales sur le modèle considéré. Enfin, le *Communication Protocol* (initialement réalisé par les *Domain-Specific Events* dans [19]) spécifie les liens entre les *Execution Functions* et les déclencheurs du *MoC* (*MoCTriggers*, les *EventTypes* dans le cas d'une *EventType Structure*). Ceci permet, en particulier, de définir comment, à l'exécution, l'ordre partiel sur les évènements du *MoCApplication* est utilisé pour orchestrer les appels aux *Execution Functions*.

Après avoir spécifié ces préoccupations, une phase de traduction est utilisée pour générer, à partir d'un modèle conforme au langage, les artefacts de niveau modèle. Ces artefacts correspondent aux spécifications du niveau langage, mais spécialisées pour le modèle considéré. Le *MoCMapping* donne donc le *MoCApplication*, les *Semantic Rules* donnent les *Semantic Rules Calls* et le *Communication Protocol* donne le *Communication Protocol Application*. Chaque préoccupation fournit le composant en charge de l'interprétation d'une spécification de niveau modèle: *Solver* (pour le *MoCApplication*), *Executor* (pour les *Semantic Rules Calls*) et *Matcher* (pour le *Communication Protocol Application*). Le composant en charge de l'exécution globale (c'est-à-dire, de coordonner les autres composants) est appelé le moteur d'exécution (*Execution Engine*). La réalisation d'un pas d'exécution du modèle se déroule ensuite de la manière suivante. Le *Solver* fournit un ensemble de solutions pour le pas courant, en conformité avec l'ordre partiel établi par le *MoCApplication*. Ces solutions sont appelées *Scheduling Solutions*. Il peut n'y en avoir aucune (situation d'interblocage), ou bien une seule, mais en général il y en a plusieurs, surtout en présence d'indéterminisme (dû par exemple à une situation de concurrence). L'une de ces solutions est sélectionnée par une heuristique du moteur d'exécution. Elle peut consister à demander à l'utilisateur d'en choisir une, à travers une interface graphique, ou à attendre qu'un programme externe choisisse, à travers une interface de programmation (*Application Programming Interface – API*). Le moteur fait ensuite appel au *Matcher* pour déterminer quels sont les *Execution Function Calls* correspondant à cette solution (en faisant correspondre les occurrences d'évènements contenues dans la solution sélectionnée avec ce qui est spécifié dans le *Communication Protocol Application*). Ces *Execution Function Calls* correspondent à des appels d'opération, qui sont donc effectués à l'aide de l'*Executor*. En conséquence de quoi, l'état courant du modèle change, correspondant bien à un pas d'exécution du modèle.

Dans la suite du chapitre, nous identifions les contraintes et limitations de l'état actuel de cette approche, pour lesquelles nous proposons ensuite des solutions. Chaque aspect est abordé en illustrant et en expliquant d'abord son intérêt ; puis en identifiant les dif-

ficultés de sa mise en oeuvre durant les phases de spécification et d'éxécution ; et enfin en présentant notre solution et ses éventuels inconvénients et coûts associés. Nous abordons par exemple le problème des règles sémantiques qui nécessitent beaucoup de temps pour s'exécuter et ralentissent donc l'exécution globale d'un modèle ; le problème des constructions de langage dont l'exécution dépend de données connues dans le modèle à l'exécution ; l'implémentation et la gestion des points de variation sémantique ; la conception de *concurrency-aware xDSMLs* visant la spécification de systèmes dits réactifs (dont le comportement est une réaction à un environnement extérieur) ; ou bien la considération du *Communication Protocol* comme interface comportementale du *concurrency-aware xDSML* (pour permettre son utilisation par d'autres langages ou d'autres programmes) et les implications quant à sa conception. Toutes ces améliorations visent à rendre l'approche plus utilisable, ou bien en proposant des outils pratiques pour spécifier certains types de constructions de langage, ou bien en rendant possible certaines constructions qui ne pouvaient auparavant pas être exprimées (ou en tout cas pas de façon idiomatique) en utilisant l'approche.

Pour finir, nous présentons l'implémentation de l'approche dans un atelier de développement de langages basé sur la plateforme Eclipse, le GEMOC Studio[1]. Celui-ci aggrège un certain nombre de technologies construites à l'aide de l'*Eclipse Modeling Framework (EMF)* et utilisées dans le cadre de l'approche proposée, comme Ecore pour la construction de métamodèles pour capturer la syntaxe abstraite des *xDSMLs*, Xtext pour la construction de syntaxes concrètes textuelles, etc. Le studio inclue aussi des technologies développées par les partenaires du projet ANR INS GEMOC comme Kermeta 3, le langage *Clock Constraint Specification Language (CCSL)*ou Sirius pour la construction de syntaxes concrètes graphiques. Nos contributions sont principalement concrétisées dans un nouveau métalangage appelé le *GEMOC Events Language (GEL)* utilisé pour spécifier le *Communication Protocol*, ainsi que dans l'implémentation du moteur d'exécution.

Les travaux présentés dans ce chapitre ont en partie été publiés dans la *8th ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2015) [85] et dans le *1st International Workshop on Executable Modeling* (EXE 2015) [86].

[1]http://gemoc.org/studio/

3.1 Introduction

3.1.1 Prerequisites

Designing Domain-Specific Modeling Languages usually revolves around the specification of the *syntaxes* of the language: both the abstract and concrete ones. The role of these concepts and how they can be specified have been detailed in Chapter 2. In this thesis, we focus on how to specify the execution semantics of an DSML, that is, how to attribute a behavior to language constructs and their relations. This makes DSMLs eXecutable (xDSMLs). More precisely, our focus is on the specification of the concurrency aspects of the execution semantics: the rules which describe how language constructs interact at runtime, how the parallel facilities of the execution platform can be exploited, etc.

We are not concerned with how models are obtained. It may be as a result of a transformation, or by the execution of a program written in a GPL using an appropriate Application Programming Interface (API), or simply via a concrete syntax defined for the xDSML. Any of these means is valid with regards to our approach.

We also consider that the static semantics associated with the abstract syntax of the xDSML have been defined beforehand, as it does not impact the specification of the execution semantics.

Still, in the scope of the ANR INS GEMOC Project (*cf.* Subsection 1.1.2), in which this thesis was realized, one of the objectives is the *animation* of the execution of concurrency-aware xDSMLs. In this project, this animation is realized based on a graphical concrete syntax, as it is usually the preferred concrete syntax for modeling languages. However, it has no impact on the description we give of our contributions. At best, we will use it to illustrate models and their execution in our implementation.

Finally, this thesis was realized in the technical context of what is called Model-Driven Engineering (MDE), and in particular borrows a lot of terminology from it. Readers unfamiliar with MDE should make sure to have read Section 2.5 before this chapter.

3.1.2 Illustrative Example

We will illustrate the concurrency-aware xDSML approach on a subset of the *Foundational Subset for Executable UML Models* (fUML) [116]. fUML is an executable subset of UML which specifies the behavioral semantics of Activity Diagrams. The semantics is mainly inspired from Petri Nets [107].

Figure 3.1 shows an excerpt from the Abstract Syntax of our implementation of fUML. An `Activity` is composed of nodes (`ActivityNode`) of various natures, connected by edges (`ActivityEdge`). Edges may have a guard if they are outgoing a `DecisionNode`, in which case the result of the guard is used to determine whether or not the branch may be executed. In any case, a DecisionNode can only result in *one* of its outgoing branches being executed.

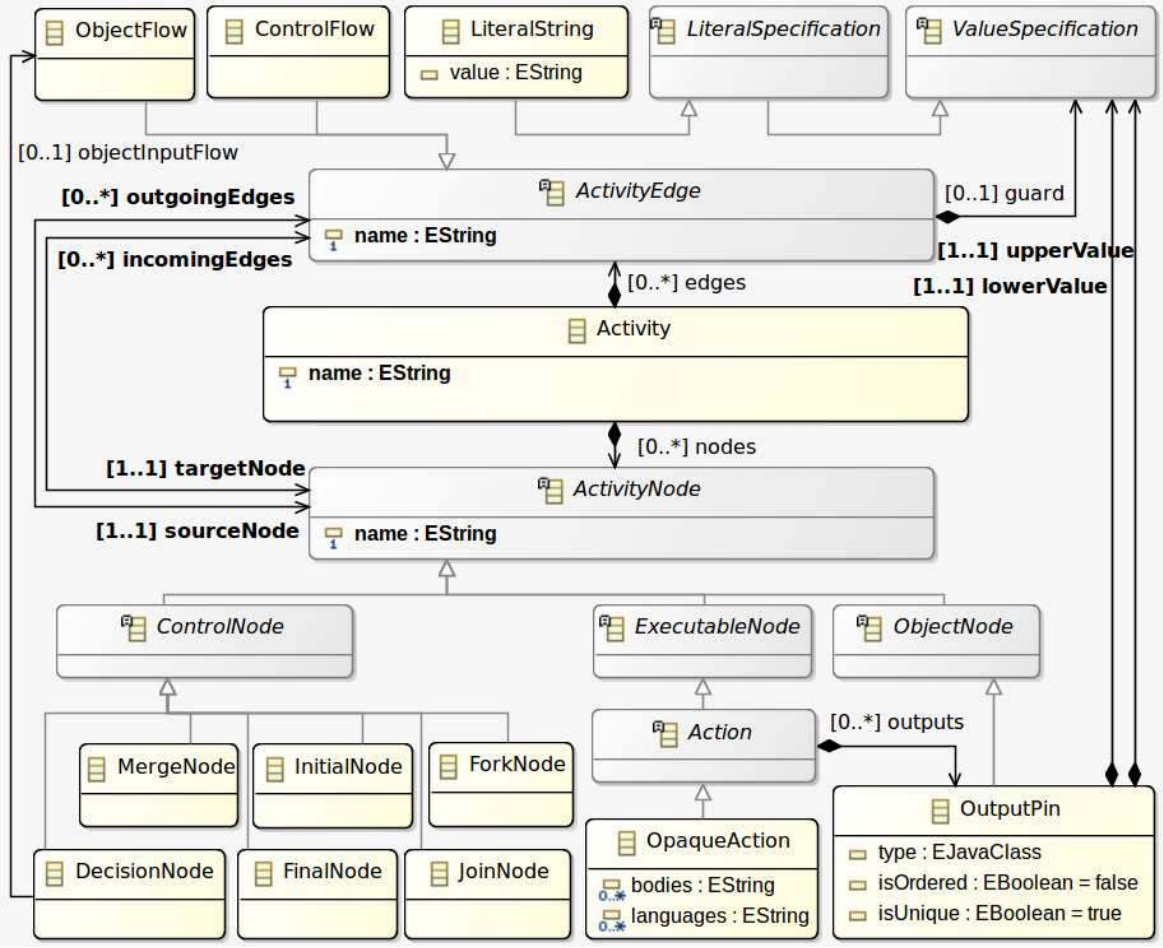

Figure 3.1: Excerpt from the fUML Abstract Syntax, presented as a meta-model.

We also need to consider an example model for this language. Figure 3.2 shows an example activity in which one drinks something while talking, for instance during a coffee break. In this Activity, the ForkNode splits the control flow into two concurrent branches. This means that the "Talk" node can be executed simultaneously with, or interleaved with, any of the nodes of the drinking part of the activity. In the drinking part of the activity, "CheckTableForDrinks" returns either "Coffee", "Tea" or "Neither". The DecisionNode

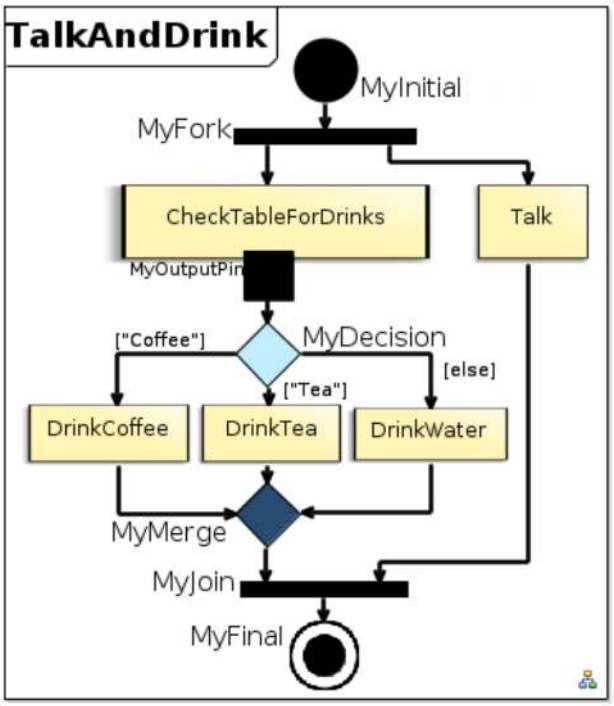

Figure 3.2: Example fUML activity where some user drinks something from the table while talking.

represents a conditional: depending on the drink found on the table, either "DrinkCoffee", "DrinkTea" or "DrinkWater" will ultimately be executed. "[else]" is the default guard in fUML, always evaluating to true but the corresponding branch can only be executed if the other branches were not possible.

3.1.3 Concurrency-aware Execution Semantics of fUML

We illustrate the initial Concurrency-aware xDSML approach, as described in [19], on fUML, and present its shortcomings.

Application of the Initial Approach

To apply the approach to fUML, we must identify in the execution semantics specification [116] which parts belong to what was called "Domain-Specific Actions" in [18, 19] and which parts belong to the concurrency concerns.

The former are the individual behaviors of each language construct. For instance, executing a node in fUML usually involves consuming incoming tokens and producing out-

going tokens. Each concrete node type does this in a slightly different manner (*e.g.*, some nodes consume without producing or vice-versa). These actions can be defined through the specification of execution functions.

The latter are the orchestration of the actions. Usually, the execution of fUML is a data flow, which means that nodes are executed based on the tokens present on their incoming edges. There are however a few variations which are possible. For instance, the fUML specification is not opinionated about how the execution of the concurrent branches should be done. The only requirement is that both branches have finished executing before the corresponding JoinNode can be executed. They can thus be executed in sequence, in parallel, or in any sort of interleaving.

fUML implementations usually hard code this decision, or relying upon the underlying execution platform. The concurrency-aware approach proposes to make explicit all these possibilities using a dedicated formalism, in order to better identify them, to allow concurrency-aware analyses to be performed on the systems, to deal with semantic variants of the language, and to refine them at deployment time for a system. For instance, we may want to prune the parallel execution of branches in case of deployment of fUML to a sequential platform. The dedicated formalism used is ModHel'X rules in [18] and the symbolic Event Structure in [19].

Shortcomings

The seminal definition of the approach has some limitations, which we illustrate on fUML.

For instance, evaluating the guard of an edge outgoing a DecisionNode is a domain-specific operation, involving fUML-specific concepts, but it does not update the model. It however provides an information as to how the orchestration must be done (*i.e.*, whether or not the branch may be executed). It is not clear in [19] how this operation must be specified and how it interacts with the concurrency concerns.

It is also not clear how calls between the Domain-Specific Actions may be realized. For instance, consider an ExecutableNode with some OutputPins. Its execution can be represented either as one action or several (its execution and then executing its pins). In particular, if data must be shared between both calls, the concurrency concerns are, by definition, not able to make the data flow between both actions. The approach should formalize how such combination of actions should be realized.

The multiplicity of the association between Domain-Specific Actions and Domain-Specific Events is also not detailed. The semantics of multiple Domain-Specific Events mapped to one Domain-Specific Action, or of one Domain-Specific Event mapped to multiple Domain-Specific Actions, are not detailed.

Another issue is that the runtime described in [19] relies on the execution of actions being short, to the point where it can be considered instantaneous. However, it is possible that the execution of a particular behavior (such as evaluating a complex expression, or retrieving specific data) takes some time to perform. So far, the runtime is sequential, which means that during the execution of such an action, other actions cannot be executed. Instead, the approach could formalize a way to perform such actions in a concurrent manner (and particularly, in parallel, if the platform used for simulation allows it). It may however have an influence on the execution flow, for instance if an important piece of data (*e.g.*, the result of a guard evaluation) conditions the future of the execution. When using xDSMLs for the purpose of *simulations* (rather than for production-grade executions), this issue is minor in the sense that it only affects the flow of the simulation (which may be mildly frustrating, but not critical). Still, we strive to make our approach as applicable as possible, so these concerns must be taken into account.

In the next section, we will formalize the description and architecture of the approach to clearly lay down the core elements of the concurrency-aware xDSMLs approach. Based on this presentation, we will then refine the shortcomings and propose features to handle them during the design of xDSMLs.

3.2 Formalization of the Concurrency-aware Approach

As explained in Section 2.2, the execution semantics of a language consists in the Semantic Mapping between the language's Abstract Syntax and its Semantic Domain (the set of all possible meanings). In the concurrency-aware approach, the concerns of the execution semantics are separated. The formalization of this separation presented hereafter results from this thesis's contributions to the approach, so the names of the concerns have been updated (compared to those used in [18, 19]) to better reflect their responsibilities. The data and its operations are gathered in the *Semantic Rules* (formerly "Domain-Specific Actions"), while the concurrency concerns are captured as the *Model of Concurrency Mapping*. Both specifications are connected by a *Communication Protocol* (included what was denoted as "Domain-Specific Events"). Figure 3.3 shows the general idea of this separation of concerns.

Our approach takes place at the language-level (*i.e.*, we specify languages) but the runtime of the specifications takes place at the model-level (*i.e.*, similar to how, in Object-Oriented Programming, instance methods defined in a class are applied for an object instance of that class). For each concern, we will explain how the model-level specification is obtained for a given model. In particular, we will often designate by "concurrency model"

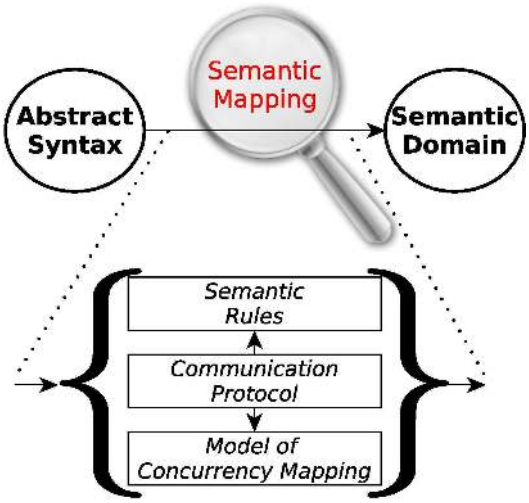

Figure 3.3: Separation of the concerns of the Semantic Mapping.

both the MoCMapping (specification at the language-level) and its model-level counterpart (which is the one used at runtime for a given model).

3.2.1 Semantic Rules

This notion was originally introduced in the Executable DSML Pattern [17], and adapted in [18, 19]. The Semantic Rules are composed of two parts.

First, the *Execution Data* capture the runtime state of a model during its execution, *e.g.*, the value of a variable, the current state of a state machine, the number of tokens in a place, etc. In fUML, edges carry `Tokens` which may be of two natures (control or data).

The second part is the *Execution Functions* which specify how the Execution Data evolve at runtime. For instance, a node in fUML can be executed, resulting in changes in the tokens held by its incoming and outgoing edges.

Figure 3.4 shows the structure of the Semantic Rules as a metamodel. Execution Data are defined in the context of a concept from the Abstract Syntax of the language (represented by the `AbstractSyntaxConcept` type). The body of an Execution Function is represented as the EOperation `behavior()`.

Figure 3.5 shows the Semantic Rules of fUML as a metamodel extending the Abstract Syntax of fUML, while Listing 3.1 shows an example implementation, using pseudo-code, of an Execution Function of fUML.

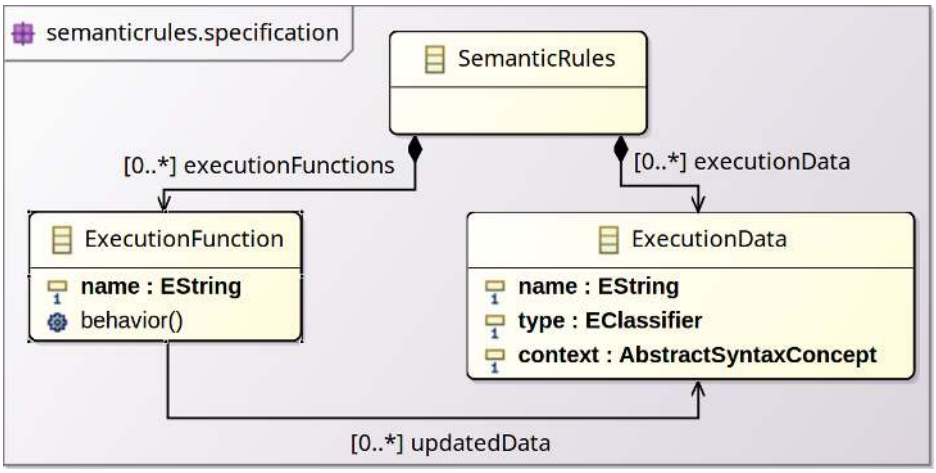

Figure 3.4: Metamodel representing the structure of the Semantic Rules of an xDSML.

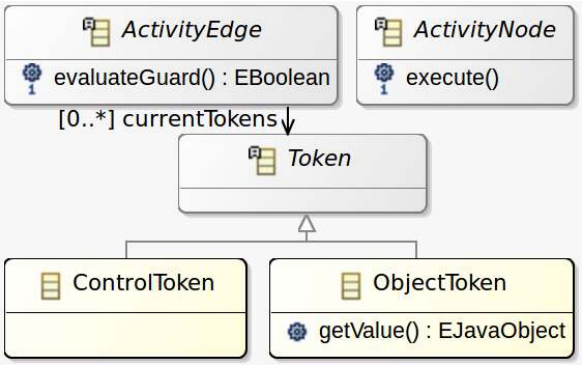

Figure 3.5: Semantic Rules of fUML as a metamodel extending the Abstract Syntax.

Listing 3.1: Implementation of an fUML Execution Function, specified using pseudo-code.

```
context ForkNode:
  def void execute():
    self.outgoingEdges.forEach[ outgoingEdge |
      self.incomingEdges.forEach[ incomingEdge |
        incomingEdge.currentTokens.forEach[ token |
          outgoingEdge.currentTokens.add(token.copy())
        ]
      ]
    ]
    self.incomingEdges.forEach[ incomingEdge |
     incomingEdge.currentTokens.clear()
    ]
```

3.2.2 Model of Concurrency Mapping

The Model of Concurrency Mapping (MoCMapping) specifies the systematic use of a MoC for the xDSML being developed. Thus, for any model conforming to the Abstract Syntax of the language, the MoCMapping is used to generate a corresponding Model of Concurrency Application (MoCApplication). The MoCApplication is a "program" in itself, conforming to the MoC used, which represents the concurrency concerns of the model.

The initial concurrency-aware xDSML approach relies on the Event Structures [160] Model of Concurrency[2]. Consequently, the MoCMapping is a specification of how, for a model, its corresponding Event Structure is obtained. The formalism used for the MoCMapping in that case is called *EventType Structures*.

Figure 3.6 recapitulates the relations between the different specifications pertaining to the concurrency concerns of a concurrency-aware xDSML or of an executable model.

Figure 3.7 shows the metamodel for the MoCApplication and its execution. It also shows the metamodel for Event Structures and their execution, and how they implement

[2]In Chapter 4, we will present a solution to define and integrate other MoCs into the approach.

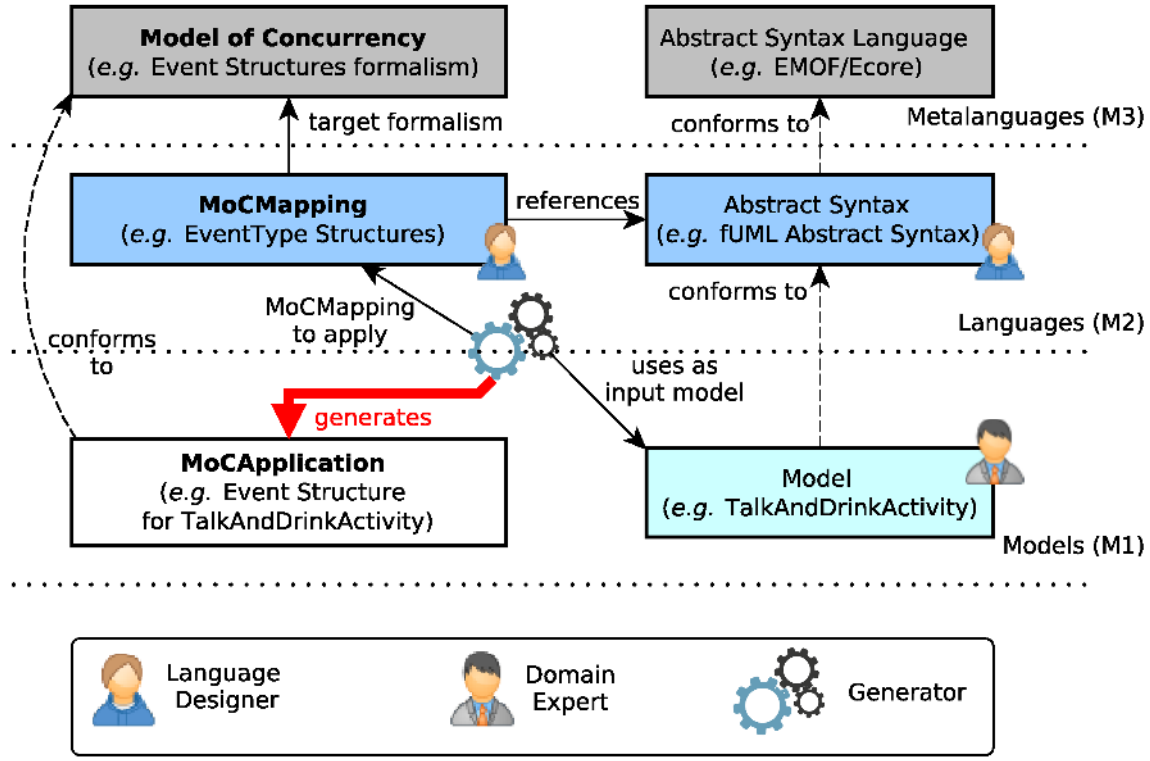

Figure 3.6: Overview of the different specifications related to the concurrency concerns of a language or model.

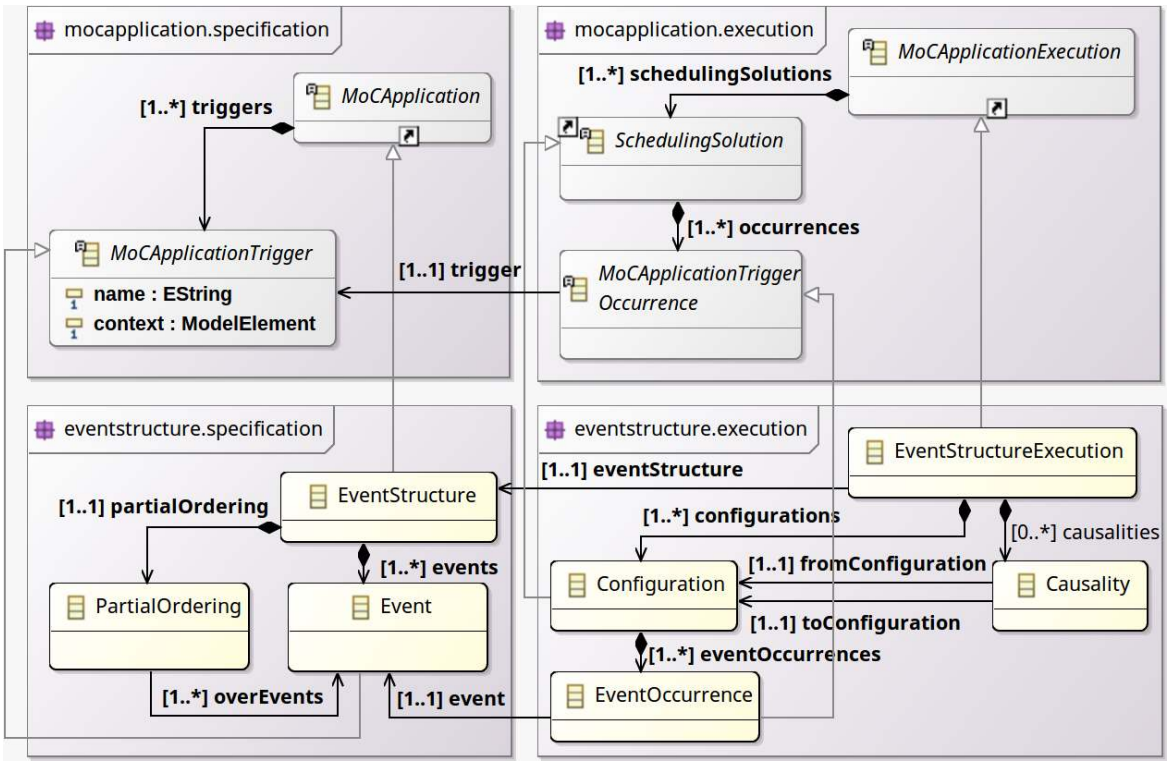

Figure 3.7: Metamodel representing the Abstract Syntax of Event Structures and their execution.

the former. The MoCApplication is made of *MoCApplicationTriggers* (*i.e.*, its stimuli, the Events in an Event Structure). Its execution is a succession of *Scheduling Solutions* containing occurrences of the MoCApplicationTriggers. In an Event Structure, the events are constrained by a partial ordering, thus specifying "when" their occurrences happen.

Let us illustrate the Event Structures MoC on fUML. Figure 3.8 shows the simplified Event Structure corresponding to our example Activity. In this graphical representation of the execution of an Event Structure, a node is a *Configuration*: an *unordered set of event occurrences* which have happened at this point. For representation purposes, "..." in a configuration represents the collection of event occurrences from the previous configurations, *e.g.*, {..., e_MyFork} is {e_MyInitial, e_MyFork}. This Event Structure captures all the possible execution paths for the model: the two concurrent branches which can be executed in parallel or interleaved, and the three different possibilities resulting of the DecisionNode.

If several execution paths are allowed at a point in the event structure, it means that there is either *Concurrency* or *Conflict*.

Concurrency means that other events are happening concurrently (interleaved or in parallel), in which case the execution paths will eventually merge. It does not mean that

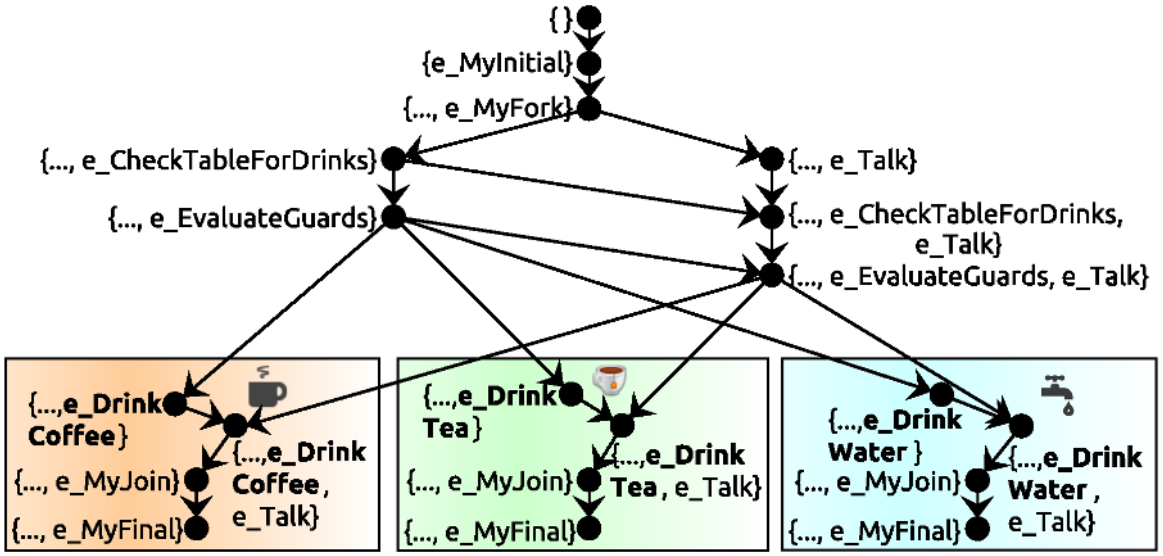

Figure 3.8: Event Structure for the fUML Activity from Figure 3.2.

the executed model reaches the same state, but instead that in terms of pure control flow (independent of any data from the model) it is at the same point in the execution. This is the case between the two branches of the ForkNode: ultimately, both branches have been executed.

Conflict means that there is a disjunction among the possible execution paths, which ultimately results in different final configurations of the event structure. Conflicts can be the sign of nondeterminism in the semantics of the language (*i.e.*, at some point, an arbitrary decision is realized). There is a conflict in the example Event Structure: the decision node leads to three different "families" of execution of the same model (one family per possible type of drink).

The MoCMapping specifies how to obtain the MoCApplication (*i.e.*, Event Structure) for any model conforming to the abstract syntax of the language. Figure 3.9 shows the metamodel for the MoCMapping. It also shows the metamodel of the EventTypes formalism, and how it implements the MoCMapping. The MoCMapping consists in a set of *MoCTriggers* which represent, at the language level, the stimuli of the MoC used. In the case of EventType Structures, the EventTypes are the MoCTriggers. In an EventType Structure, these MoCTriggers are symbolically partially ordered (by the *SymbolicPartialOrdering*), that is, there is a specification of how the model-level partial ordering is obtained. An EventType (or more generally, a MoCTrigger) is defined in the context of concepts from the Abstract Syntax of the language (represented by the `AbstractSyntaxConcept` type). The symbolic partial ordering can be specified through a set of symbolic *constraints*

over the EventTypes. When they are unfolded for a given model, it results in constraints defining the partial ordering of the Event Structure.

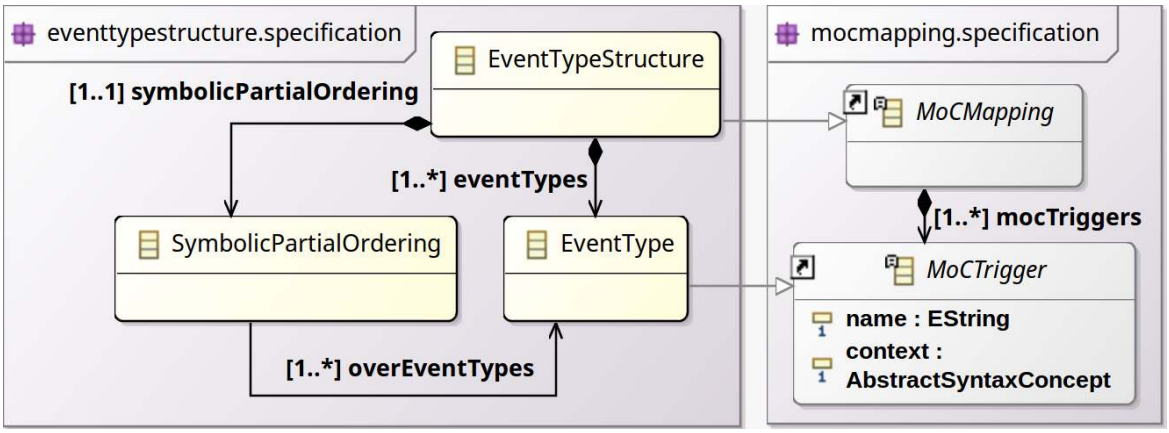

Figure 3.9: Metamodel representing the Abstract Syntax of the EventTypes formalism.

For fUML, the two main points of interest are the *execution of a node* and the *evaluation of the guard of an edge*. Figure 3.10 shows the declaration of these two EventTypes in the context of concepts from the AS of fUML.

Figure 3.10: EventTypes `executeNode` and `evaluateGuard` for fUML, declared in the context of a concept from the AS of fUML.

Then, we want to specify constraints over these EventTypes such that, for a model, the resulting Event Structure defines a partial ordering in conformance with the semantics of fUML. The main idea in fUML is that an edge's source is executed before its target. Some nodes however, are a bit different. For instance, for a JoinNode, we need to make sure all of the incoming branches have finished executing before we can execute the JoinNode. MergeNode is also peculiar, because it is the dual of DecisionNode, and is executed whenever one of the incoming branches has been executed. Specifying these constraints depends on the expressive power made available by the metalanguage used to specify EventType Structures. Listing 3.2 shows an example specification, using pseudo-code, of constraints between EventTypes.

Listing 3.2: Example constraints between EventTypes of fUML, specified using pseudo-code.

```
context ActivityEdge:
  constraint executeSourceBeforeTarget:
    if(self.guard == null and !self.targetNode kindof MergeNode) {
      self.sourceNode.executeNode
        strictly precedes self.targetNode.executeNode;
    }
```

In this example, the constraint `strictly precedes` between two events *e*_foo and *e*_bar means that the i^{th} occurrence of *e*_foo happens strictly before the i^{th} occurrence of *e*_bar. In mathematical terms, this can be formalized as:

$$\forall i \in \mathbb{N}, e\_\text{foo}_i < e\_\text{bar}_i$$

3.2.3 Communication Protocol

Finally, the Communication Protocol is in charge of matching the MoCTriggers of the MoCMapping (which represent abstract actions) with the Execution Functions of the Semantic Rules. This effectively defines how, at runtime, the MoCApplication is used to orchestrate the calls to the Execution Functions, therefore implementing the execution of a model. More formally, the Communication Protocol defines *Mappings* between a *MoCTrigger* (the EventTypes in an EventType Structure, made available by the MoCMapping) and an Execution Function. Figure 3.11 shows the metamodel representing the structure of the Communication Protocol.

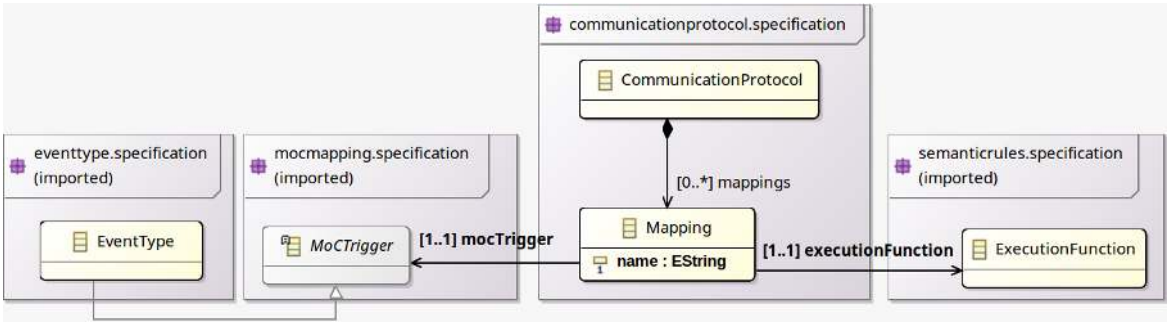

Figure 3.11: Metamodel representing the Abstract Syntax of the Communication Protocol.

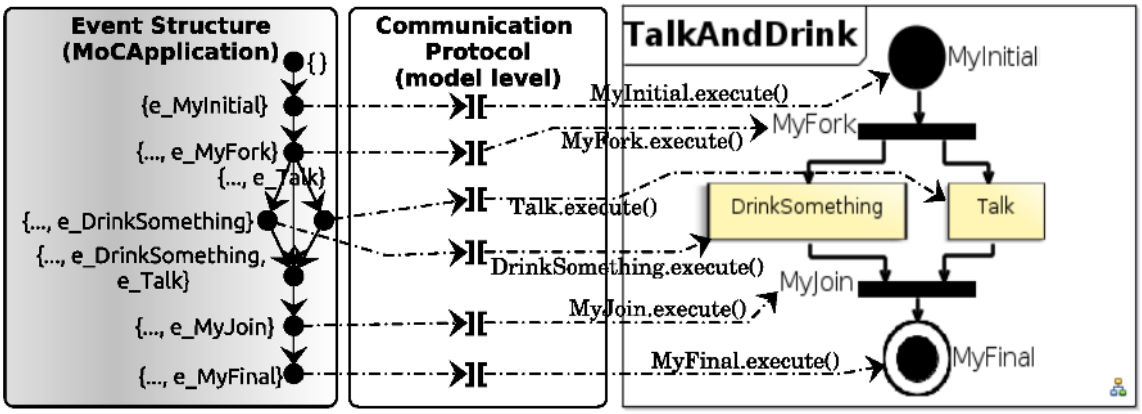

Figure 3.12: Overview of the model-level specifications for a simplified version of our example fUML activity.

The different specifications at the model-level for a simplified version of the example activity (for representation purposes) are shown on Figure 3.12. The node "DrinkSomething" represents the drinking part of the activity of Figure 3.2. In this figure, the Event Structure on the left captures all the possible execution paths of the model. After initializing the activity, the ForkNode is executed. Then, in this simplified view, there are three solutions: drinking something then talking, talking and then drinking something, or talking while drinking something. Ultimately the same configuration is attained. After that, the JoinNode and FinalNode can be executed. For this simplified activity, this gives us 3 possible scenarios in total. But for a more complicated model like the one on Figure 3.2, we have a total of 64 different possible scenarios, accounting for all the possible interleavings and parallelisms between the talking and drinking part of the activity, and the different possible orders of evaluation of the guards. See Appendix A for the detail of all the possible execution scenarios.

Listing 3.3 shows an excerpt from the Communication Protocol of fUML specified using pseudo-code. There are two mappings, one for the execution of nodes, and one of the evaluation of the guards of edges.

Listing 3.3: Excerpt from the Communication Protocol of fUML, specified using pseudo-code.

```
// Syntax:
// Mapping [mapping name]:
//   upon [MoCTrigger from MoCMapping]
//   triggers [Execution Function from Semantic Rules]

Mapping ExecuteActivityNode:
  upon executeNode
  triggers ActivityNode.execute()

Mapping EvaluateGuard:
  upon evaluateGuard
  triggers ActivityEdge.evaluateGuard()
```

3.2.4 Generation of the Model-level Specifications

The execution semantics are defined at the language level, but they are applied when executing a particular model. Therefore to facilitate the development and debugging of our approach, we first "unfold" the execution semantics specification for a particular model. Figure 3.13 sums up the generation of the three concerns.

- **1**: Model + Semantic Rules → **Semantic Rules Calls**
  Captures the dynamic data of the model and the API that makes these data evolve during runtime.

- **2**: Model + MoCMapping → **Model of Concurrency Application**
  Partial ordering over abstract events (*i.e.*, more formally, the MoCApplicationTriggers, *e.g.*, the events of an Event Structure), independent of any data from the model. See Figure 3.8 for the simplified Event Structure corresponding to the MoCApplication of the example Activity.

- **3**: Model + Semantic Rules Calls + MoCApplication + Communication Protocol → **Communication Protocol Application**
  Mappings (called *MappingApplications* to differentiate them from their language-level counterparts) between the abstract events from the MoCApplication and Execution Function calls.

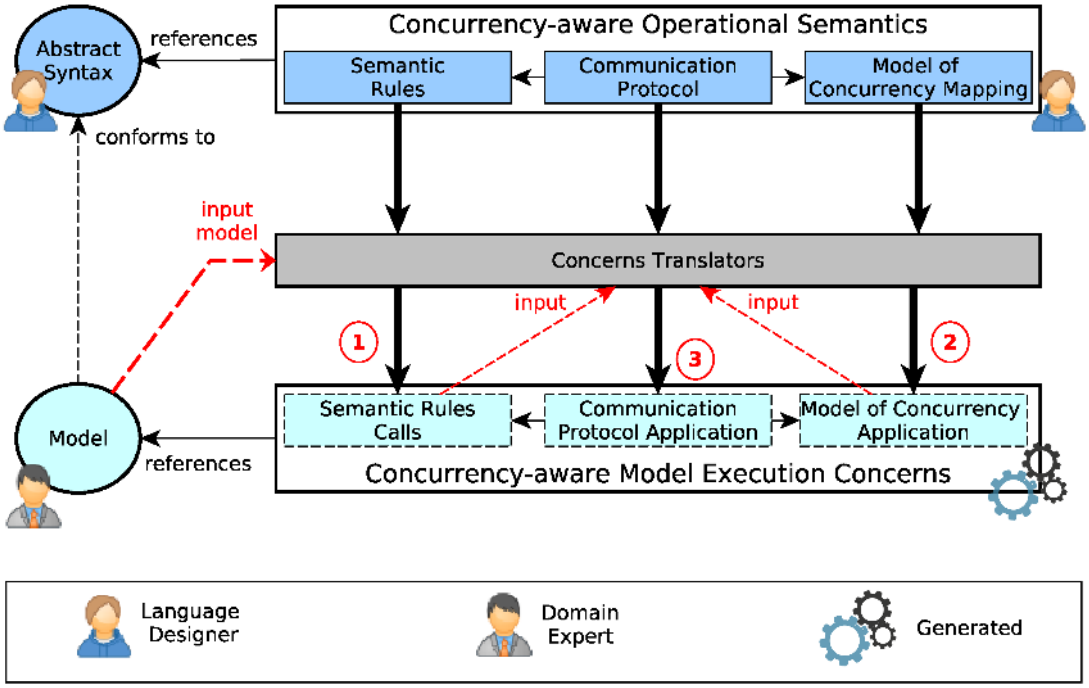

Figure 3.13: Generation of the different model-level concerns

As an example, Listing 3.4 shows an excerpt from the Communication Protocol Application for our example fUML Activity. It shows the pseudo-code specification obtained by automatically unfolding the Communication Protocol shown on Listing 3.3 for the Activity shown on Figure 3.2. This specification is often big and repetitive since models often have several instances of the same concept from the abstract syntax, so the unfolding of the Communication Protocol results in a same specification being adapted for each instance.

Listing 3.4: Excerpt from the generated Communication Protocol Application for the example fUML Activity, specified using pseudo-code.

```
1  // Syntax:
2  // MappingApplication [mapping application name]:
3  //   upon [MoCApplicationTrigger from MoCApplication]
4  //   triggers [Execution Function Call from Semantic Rules Calls]
5
6  MappingApplication ExecuteActivityNode_MyInitial:
7    upon executeNode_MyInitial
8    triggers MyInitial.execute()
9
10 MappingApplication ExecuteActivityNode_MyFork:
11   upon executeNode_MyFork
```

```
12    triggers MyMyFork.execute()
13
14  MappingApplication ExecuteActivityNode_Talk:
15    upon executeNode_Talk
16    triggers Talk.execute()
17
18  MappingApplication ExecuteActivityNode_CheckTableForDrinks:
19    upon executeNode_CheckTableForDrinks
20    triggers CheckTableForDrinks.execute()
21
22  MappingApplication ExecuteActivityNode_MyOutputPin:
23    upon executeNode_MyOutputPin
24    triggers MyOutputPin.execute()
25
26  MappingApplication ExecuteActivityNode_MyDecision:
27    upon executeNode_MyDecision
28    triggers MyDecision.execute()
29
30  MappingApplication ExecuteActivityNode_DrinkCoffee:
31    upon executeNode_DrinkCoffee
32    triggers DrinkCoffee.execute()
33
34  MappingApplication ExecuteActivityNode_DrinkTea:
35    upon executeNode_DrinkTea
36    triggers DrinkTea.execute()
37
38  MappingApplication ExecuteActivityNode_DrinkWater:
39    upon executeNode_DrinkWater
40    triggers DrinkWater.execute()
41
42  MappingApplication ExecuteActivityNode_MyMerge:
43    upon executeNode_MyMerge
44    triggers MyMerge.execute()
45
46  MappingApplication ExecuteActivityNode_MyJoin:
47    upon executeNode_MyJoin
48    triggers MyJoin.execute()
49
50  MappingApplication ExecuteActivityNode_MyFinal:
51    upon executeNode_MyFinal
52    triggers MyFinal.execute()
53
54  MappingApplication EvaluateGuard_MyDecision2DrinkCoffee:
55    upon evaluateGuard_MyDecision2DrinkCoffee
```

```
56    triggers MyDecision2DrinkCoffee.evaluateGuard()
57
58  MappingApplication EvaluateGuard_MyDecision2DrinkTea:
59    upon evaluateGuard_MyDecision2DrinkTea
60    triggers MyDecision2DrinkTea.evaluateGuard()
61
62  MappingApplication EvaluateGuard_MyDecision2DrinkWater:
63    upon evaluateGuard_MyDecision2DrinkWater
64    triggers MyDecision2DrinkWater.evaluateGuard()
```

3.2.5 Runtime

Each concern has an associated runtime: **Solver** for the MoCApplication, **Executor** for the Semantic Rules Calls and **Matcher** for the Communication Protocol Application. These runtimes are coordinated by the runtime for the whole language called the **Execution Engine**.

An overview of the architecture of the runtime is shown on Figure 3.14. It is totally generic (*i.e.*, agnostic of the technologies and tools used for each concern) thanks to the use of an inversion of control mechanism (*e.g.*, dependency injection).

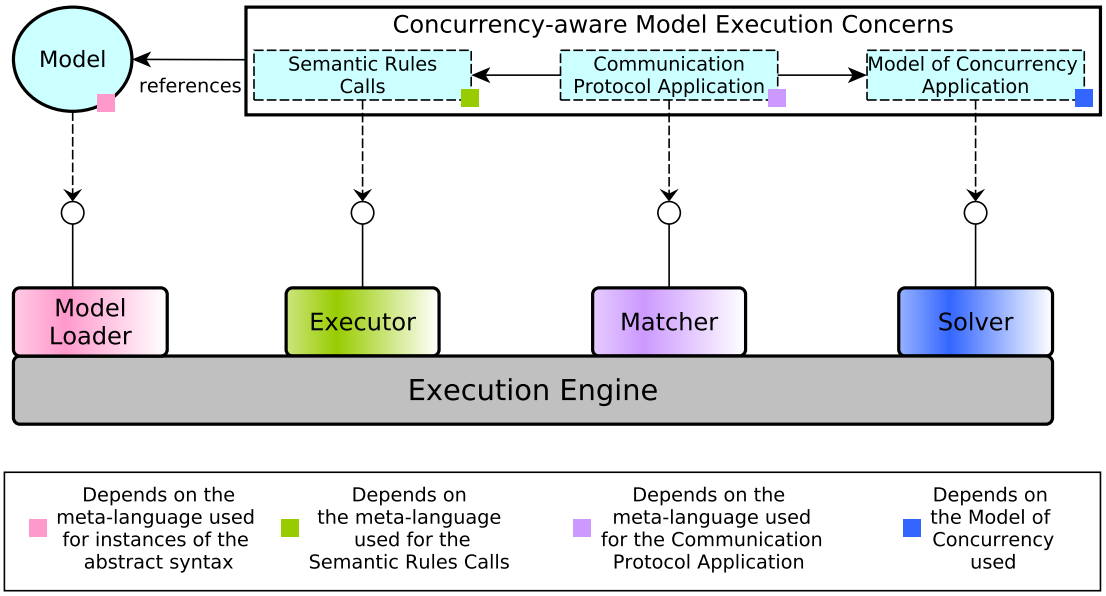

Figure 3.14: Architecture of the runtime of a concurrency-aware xDSML

Figure 3.15 shows the simplified sequence diagram corresponding to the realization of one step of execution. First, the Solver provides the set of possible solutions, called

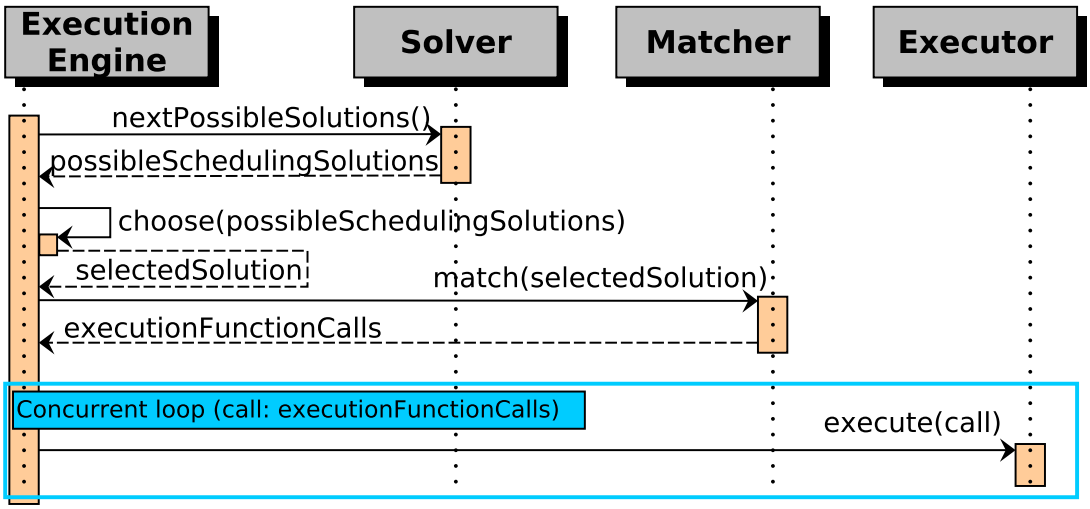

Figure 3.15: Sequence Diagram of a step of execution

Scheduling Solutions. There may be no solution, as a consequence of a deadlock (possibly because the execution has finished). There may also be only one possible, but most often when there is nondeterminism such as in case of concurrency, several are possible. One of these solutions is chosen via an heuristic of the runtime, which can in particular consist in asking the user to choose one solution through a UI (i.e., in case of step-by-step execution). It then uses the Matcher to retrieve the corresponding set of Execution Function calls. These are executed thanks to the Executor in a concurrent mode (*e.g.*, in any order, in paralle, etc.) as they are considered as happening simultaneously (with regards to the MoCApplication). This effectively triggers changes in the Execution Data, thus corresponding to making the model evolve due to its execution.

3.2.6 Refinement of the Shortcomings

The concurrency-aware approach we have defined so far has some limitations which we intend to overcome. Some of them were illustrated on fUML in 3.1.3. We propose to identify them based on the modular design we just presented. When considering solutions to these shortcomings, we will strive to respect the following constraints.

1. To keep intact the initial objectives of the approach regarding the modularity and analyzability of the semantics. This means that the separation between the data aspects of the semantics, and the concurrent aspects, must be respected.

2. To not rely on modifying the MoC or its runtime. The purpose of the approach is to be able to analyze the MoCApplication using existing tools and methodologies available for the MoC used.

3. To make the implementation of the Semantic Rules as idiomatic as possible. This activity is the closest to traditional programming, and should therefore have an adequate syntax.

Refinement of the Design of the Semantic Rules

The Semantic Rules capture the dynamic data of models and how they evolve. Due to their nature, their specification is very close to traditional programming activities: specifying data and operations (or algorithms) exploiting these data. This makes Turing-complete, or by extension, GPLs, good candidates as metalanguages for the Semantic Rules. But our approach relies on a clear design of the Semantic Rules, therefore some programming features possibly brought by the chosen metalanguage must not be used. We detail these issues in Section 3.3.

Non-blocking Execution Function Calls

During the execution, Execution Function calls are realized in a blocking way. This means that once they have been scheduled and their execution starts, the rest of the execution is on hold. This is fine for most Execution Functions which should generally manipulate data available in the model, and whose execution time is short enough to be neglected. But this is an issue if the Execution Function is supposed to do heavy computations, access a lot of data, retrieve external resources, or connect to some network. This disrupts the rest of the execution, even the parts which are not dependent on the results of the time-taking operation. Therefore, we explore the issue of running Execution Function calls in a non-blocking manner and its limitations in Section 3.4.

Improving the Communication Protocol to Deal with the Completion of Execution Function Calls

So far, the Communication Protocol is a one-way communication from the MoCMapping, to the Semantic Rules. However, for some language constructs, the control flow depends on data returned by a Query at runtime. For instance, in fUML, after a DecisionNode, one of its branches is executed depending on the results of the guard evaluations. It may also depend on some previously-called (non-blocking) Execution Function call being finished. We propose to enrich the Communication Protocol with the means to specify these kinds of

communications. Sections 3.5 and 3.6 illustrate the challenges and our solutions to specify these communications.

Reuse of Execution Functions

Execution Functions are designed with the intent of being called by the Execution Engine because the MoCApplication orchestrated its call. In programming languages, operations (or procedures, functions, etc.) are used to share code, which means that an operation is usually called from several different points of a program. In particular, an Execution Function implementation may want to rely on the use of another Execution Function, either to avoid duplicating code, or simply to gain access to a particular piece of data. If we want to be able to maintain the concurrency-awareness of this internal use of another Execution Function, then an adequate coordination between the semantics concerns must be specified. In Section 3.7 we identify the difficulties bound to this issue, propose elements of solution towards the specification of composite Execution Functions and show the limitations of this feature.

Semantic Variation Points

The concurrency-aware approach modularizes the semantics specification of an xDSML, thus making possible the variability of some of the parts of the specification. Language specifications sometimes include Semantic Variation Points (SVPs) in order to leave implementors and users with some degree of freedom to adapt the language to specific situations. In Section 3.8 we discuss how SVPs can be specified and implemented in the approach. In particular, we show that the concurrency concerns specification eases the implementation of SVPs related to the concurrency of a language.

Concurrency-aware xDSMLs for Reactive Systems

One of the constraints of the separation of concerns is that the MoCMapping is data-independent. This means that data flows can be difficult to implement, both between Execution Functions (which leads to additional difficulties treated in Section 3.7) and from a component external to an Execution Function. This means that reactive systems are, so far, difficult to capture using concurrency-aware xDSMLs. We propose in Section 3.9 a means to enable the specification of languages in which data flows can be realized, facilitating the design of reactive systems.

Behavioral Interface of Concurrency-aware xDSMLs

The Mappings of the Communication Protocol represent the interface for the behavior of individual elements of the language which, put together, represent the whole behavior of the language. This interface can be exploited by various components: a Graphical User Interface (GUI) to implement the heuristic of the runtime, the runtime of another language (possibly concurrency-aware), a trace that records which Mappings have occurred during the execution, etc. To accommodate the variety of needs from these external components, we propose in Section 3.10 elements of solutions to improve or refine the interface presented by the Mappings of the xDSML. We also identify some associated issues and limitations.

Tailoring the Model of Concurrency used to the Concurrency Paradigm of the xDSML Under Development

Finally, the main issue with the approach we have presented so far is that the only available Model of Concurrency is Event Structures. This is mainly due to the complexity of the design the metalanguage used to specify the MoCMapping. Traditionally, MoCs are used directly at the model level, in which case the mapping between a MoC and the AS of an xDSML is not developed. Even if we were to add a new MoC to our approach, it would need to be formatted in a particular way to fit our approach. In Chapter 4 we will show how to use concurrency-aware xDSMLs as MoCs, thus providing an effortless means to use new formalisms to capture the concurrency concerns of xDSMLs.

3.3 Refining the Design of the Semantic Rules

We make explicit and present some design constraints for the Semantic Rules to be consistent with our approach. We also refine the role of the Execution Data and Execution Functions.

3.3.1 Exploiting the Execution Data

The Execution Data define the set of dynamic data that evolve during the execution of a model. Their only focus is to represent the pure execution of models. Additional layers may be specified on top of that for specific purposes.

For instance, in the context of representing the execution of models, the difference must be drawn between the Execution Data and their formatting for an animation representation. We call the latter the *Animation Data*. They define a particular point of view

on the Execution Data, which will be used to animate (textually, graphically or other) the executed model. The difference between the Execution and Animation Data is a bit similar to the difference, in Object-Oriented Programming, between a class's fields (internal representation of the data held by a class) and its public accessors (its interface with other classes).

In fUML, the Execution Data are mainly the Tokens held by the edges. But in a graphical animation of the execution of fUML, representing the tokens might not be the most attractive representation. Instead, we can prefer to represent which nodes may be executed (computed based on the tokens on the incoming edges of the nodes). Identifying which nodes may be executed is a view on the Execution Data of fUML, which is defined at the animation layer level and should not be done in the Semantic Rules. For technical reasons, one may need to define the Animation Data alongside the Execution Data if the animation layer used does not provide adequate means for their definition.

Other layers above the Execution Data may be considered, for instance if we want to perform some form of analyses on the runtime state of models during their execution. In any case, the definition of the Execution Data should not be polluted by these external concerns.

3.3.2 Taxonomy of Execution Functions

We have identified in our approach two natures of Execution Functions: *Modifiers* and *Queries*. The difference is mainly conceptual but could be concretized in the Semantic Rules metalanguage, although it would require significant design effort. It can be left as a methodological aspect of our approach. Figure 3.16 shows the updated metamodel for the Semantic Rules with the taxonomy we propose.

Modifiers

Modifiers are functions with side-effects, whose role is to update the runtime state of the model when executed. For instance in fUML, when a node is executed, it modifies the runtime state of the incoming and outgoing edges (the tokens they hold). Figure 3.17 illustrates the impact of the execution of `MyForkNode` on its incoming and outgoing edges.

Queries

Queries are side-effect-free functions whose role is to return runtime information, either about the model itself or computed based on data from the model. In fUML, evaluating

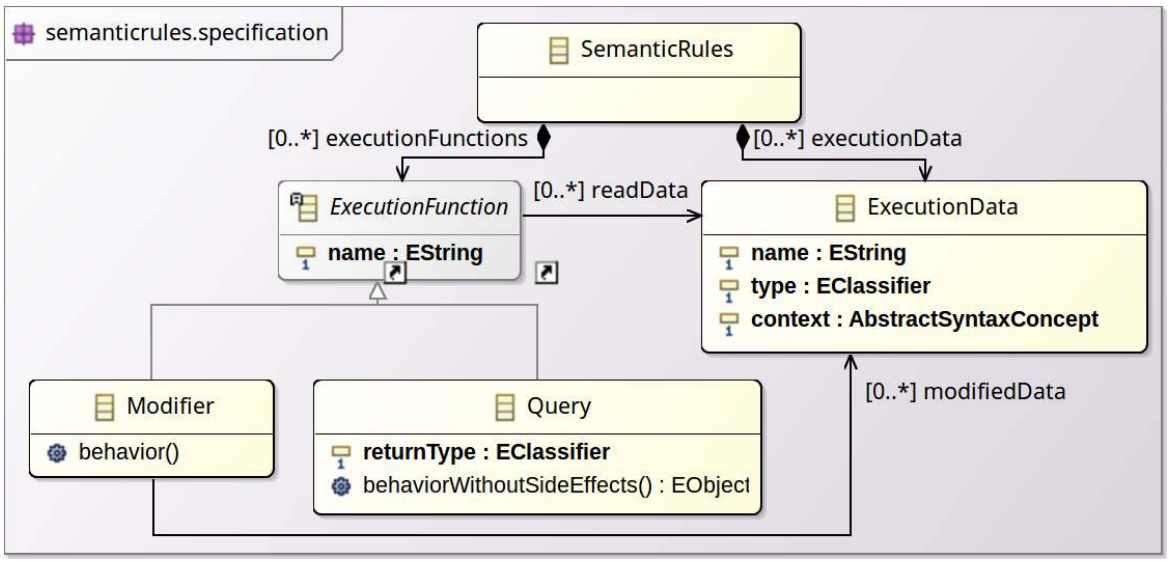

Figure 3.16: Metamodel of the Semantic Rules showing the taxonomy of Execution Functions.

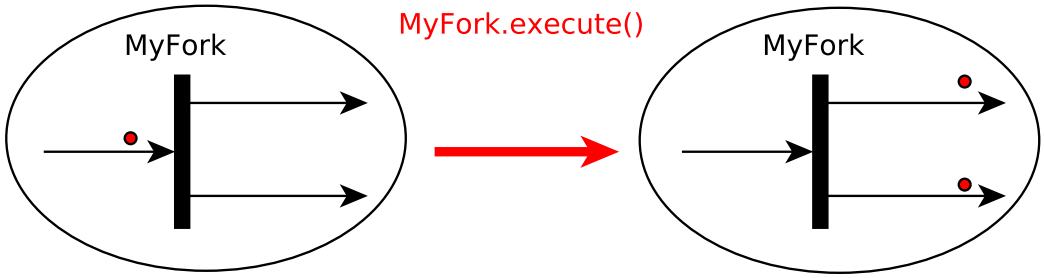

Figure 3.17: Example Modifier: `ActivityNode.execute()` modifies the tokens held by incoming and outgoing edges.

the guard of an edge is a Query which returns a boolean value. Figure 3.18 illustrates this query.

3.3.3 Depth of the Concurrency-awareness

A key consideration in the design of the Execution Functions is that they represent the interface making explicit how the Execution Data evolve. They are the point of contact for the rest of the semantics. The particular operations they realize in their body are not visible individually for the rest of the semantics. This means that the concurrency model only captures the concurrency between the Execution Functions calls ; it cannot go “deeper”, such as inside the body of an Execution Function.

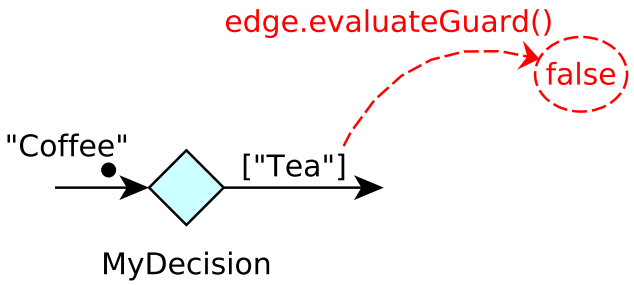

Figure 3.18: Example Query: `ActivityEdge.evaluateGuard() : Boolean` returns whether or not a branch may be executed.

This is usually the case when using any Model of Concurrency. The MoC is used to help schedule some "atomic" actions which are not themselves decomposed using the MoC. In the concurrency-aware approach, from the point of view of the concurrency model, the body of Execution Functions is "opaque", *i.e.*, it cannot be seen and thus is not explicitly scheduled. Instead, it follows the control flow of the metalanguage used to specify the Semantic Rules. Placing the atomicity of the concurrency model at the Execution Function level allows the approach to remain open to any metalanguage for the Semantic Rules, including ones where complex data operations can be performed. This is the case for instance if Java is used to implement the Execution Functions as methods.

To illustrate this concept, let us consider the evaluation of binary expressions such as $a + b$, where a and b are expressions. Evaluating such an expression consists in first evaluating a and b, and then summing their results. This can be done in many ways, most commonly either first computing a, then b; first computing b and then a; or possibly computing both in parallel. Using the concurrency-aware approach for this case, these variations can be captured in two different manners. Either they are made explicit in the concurrency model, as prescribed by the approach, enabling their analysis but requiring dedicated specifications (*cf.* the description of the approach); or they are made implicit in the Semantic Rules, relying on metalanguage-specific primitives, and hindering any possible use of concurrency-aware analyses.

We show the difference between these two approaches in the following figures. On Figure 3.19, we show the execution concerns for the computation of this expression, in the case where the concurrency-awareness is not very deep, *i.e.*, it does not detail in which order a and b are computed. Instead, this is left to the implementation of the Execution Function `Expression.evaluate()`. Overall, this is very much like what happens when defining an interpreter for expressions in a traditional manner (*e.g.*, with the Visitor design pattern).

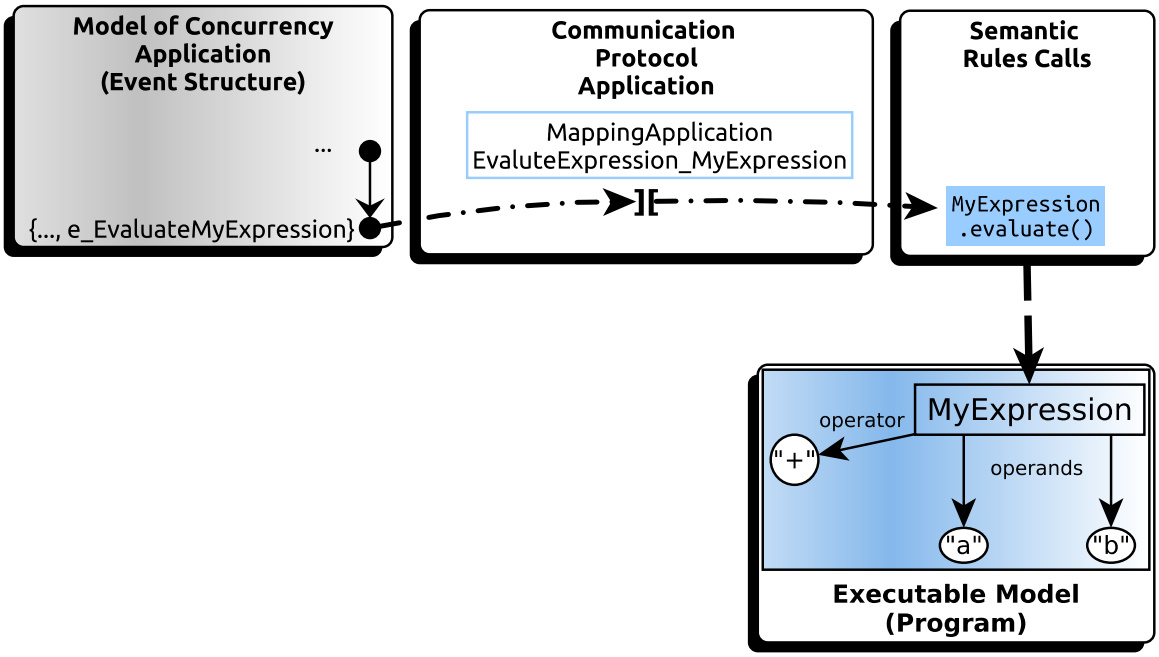

Figure 3.19: Execution concerns for the execution of expression $a + b$, where the concurrency aspects are not detailed in the concurrency model.

On Figure 3.20, these concerns are made explicit in the concurrency model, as described in the approach. This requires additional efforts (*i.e.*, compared to traditional approaches) but allows its analysis and refinement.

Ultimately, both solutions capture the same semantics, but with a different degree of concurrency-awareness. We advocated the use of a detailed concurrency-awareness, and thus study some features to facilitate its specification and execution. However, at times, and for practical reasons, one may opt not to pay the cost of concurrency-awareness (in terms of difficulty to specify), because the benefits it provides are not deemed worthy (*e.g.*, if we want to focus on the concurrency aspects of only parts of a system). This remains a matter of appreciation from the language designer.

3.3.4 Compatibility between the MoCMapping and the Semantic Rules

The modularity of the concurrency-aware approach means that for a language, we can change its MoCMapping or Semantic Rules. But not all MoCMappings are compatible with all Semantic Rules, and vice-versa.

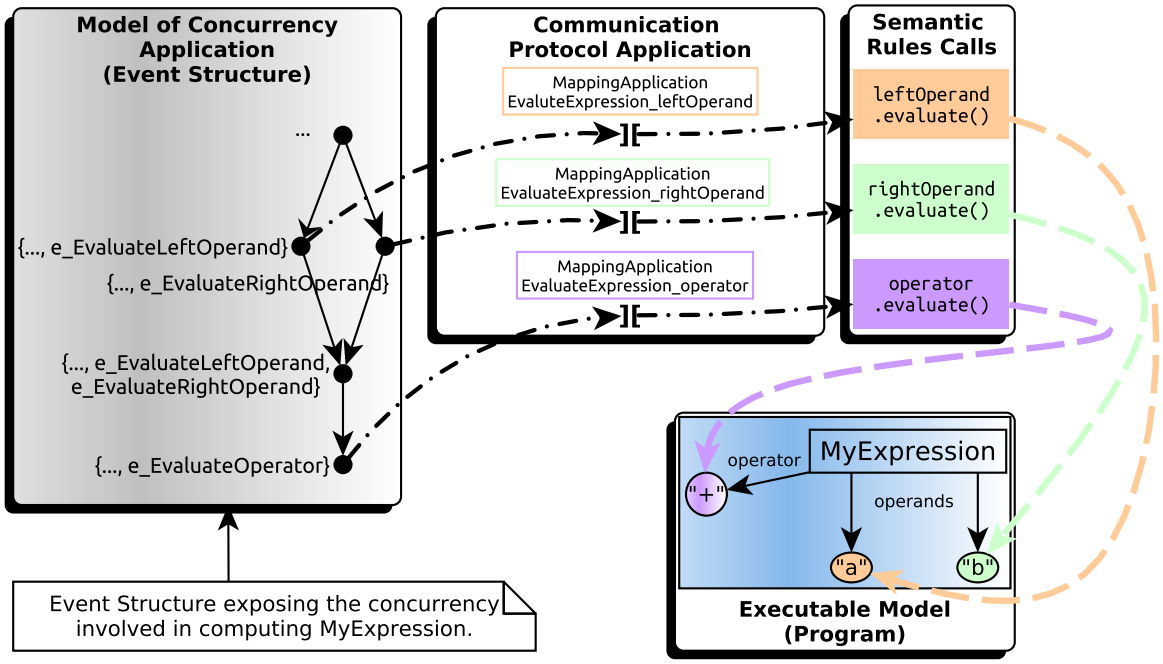

Figure 3.20: Execution concerns for the execution of expression $a + b$, where the concurrency aspects are detailed in the concurrency model.

Pre-conditions of Execution Functions

Execution Functions may be designed with a certain set of expectations with respect to the state of the model or previous operations having been performed. These requirements cannot be captured by the MoCMapping, as it is agnostic of the data from the model. It is also difficult to statically analyze the use of shared data at design time, as it would rely on analyzing the content of each Execution Function, and also relies on the intended semantics of the xDSML.

To capture these requirements, we propose to enhance Execution Functions with a set of *pre-conditions* representing the minimum requirements they expect from the runtime state of the model, before their execution can be performed. This is a common mechanism of the *design-by-contract* programming approach [97] to ensure safe interactions between software components. This mechanism is also found in modeling formalisms such as CSP [68] or Event-B [1].

An example of such pre-conditions in fUML is that we do not want to try executing a node if it does not have the required tokens on its incoming edges. This requirement can easily be captured in a pre-condition by checking the tokens present on the incoming edges.

Race Conditions

Reversely, a common issue with highly-concurrent systems is dealing with race conditions. They can be tricky to identify at design time, and are often difficult to track at runtime, as they may only happen in conditions depending on scenario-specific data, or on the underlying execution platform. In our approach, they could stem from the Communication Protocol mapping MoCTriggers to Execution Functions in such a way that two Execution Functions manipulating the same data are scheduled in parallel.

To mitigate this issue, Execution Functions should declare the model data which they read and modify. This would facilitate the identification, by the language designer, of which Execution Functions should generally not be scheduled in parallel. For instance in fUML, the Execution Function corresponding to the execution of a node reads and modifies the incoming edges' tokens and modifies the outgoing edges' tokens.

Integration into the Concurrency-aware Approach

These two mechanisms can be implemented explicitly in the Semantic Rules metalanguage. They can also be considered purely as methodological aspects, since they are mostly about guiding the language designer during the execution semantics design. As such, they can be seen as optional features whose sole purpose is to facilitate the language designer's activity.

In the latter case, the pre-conditions can be defined inside the Execution Functions implementations (possibly in their own boolean-valued function). The identification of data used by Execution Functions can be specified via annotations or even informally. Theoretically, it can be extracted automatically from the Execution Functions implementations. However, as this is a significant implementation effort, we leave open how this specification is ultimately realized, as it does not directly impact the execution of models.

3.3.5 Summary

The refinement of the Semantic Rules we have presented has two purposes. Firstly, it contributes to understanding the role of the Execution Data and Functions in the xDSML's execution semantics. Secondly, it also contributes to understanding the relation between the Semantic Rules and the MoCMapping, notably the notion of compatibility between these two aspects. This refinement also serves as the groundwork for other features presented in this thesis, and particularly in the rest of this chapter. More specifically, the role of the Execution Functions as the atomic evolutions of the language, and their taxonomy into *Queries* and *Modifiers* will be used further in this chapter.

3.4 Non-blocking Execution Function Calls

Initially, all Execution Function calls are done in a blocking manner, which means that they put the rest of the execution on hold. We propose a feature to allow Execution Function calls to be executed in a non-blocking manner. First we motivate this feature and identify its challenges. Then we propose our solution to specify these calls and the associated modifications to the runtime of concurrency-aware xDSMLs.

3.4.1 Purpose

Blocking Execution Function calls are a problem for functions which imply complex computations, access a lot of data, retrieve external resources or connect to some network. They disrupt the rest of the execution, including parts which are not dependent on their outcome.

For instance in fUML, if the execution of a node takes a long time, and that this node is on a branch between a ForkNode and a JoinNode, it would be interesting to be able to progress on the concurrent branches meanwhile. In the approach we have described so far, this is not possible: if we launch the execution of the node we must wait for it to complete before being able to do anything else. Another way to see this is to consider the execution of an Execution Function call as a couple of events: one for the beginning of the execution, and one for the end. Usually the first one depends on previous parts of the model having been executed (*e.g.*, we start the evaluation of the guards of edges outgoing a DecisionNode after the DecisionNode has been executed) ; while the second one matters for the execution of the subsequent parts of the model (*e.g.*, we execute one of the branches only when all the guards have been evaluated).

Non-blocking Execution Function calls do not fundamentally impact the representation of the concurrency concerns using a MoC, but they improve the performance of the execution by making it smoother, *i.e.*, by allowing concurrent execution of independent parts of the model. This comes at the cost of possibly making the specification of the concurrency model more complex, since new race conditions may appear as a consequence of a non-blocking execution function call. A more accurate concurrency model would thus be required to ensure these race conditions do not occur.

3.4.2 Challenges

This feature leads to the following challenges. First, we need to identify how and where the blocking/non-blocking execution strategies should be specified. Then we need to identify

how it is implemented in the runtime of concurrency-aware xDSMLs. Finally, we also need to manage the completion of non-blocking Execution Function calls in a way that is coherent with how we manage the completion of regular Execution Function calls.

3.4.3 Solution

Here we present our solution, with a focus on each associated challenge.

Specification of Non-blocking Execution Function Calls

Drawing from the experience of GPLs, where many different tactics are provided by (standard) libraries to implement non-blocking function calls ("Asynchronous Programming", *cf.* Chapter 2), we have identified two possible solutions. We can specify the blocking/non-blocking nature of the call either at the Execution Function level (in the Semantic Rules), or at the Mapping level (in the Communication Protocol).

The upside of the first solution is that the non-blocking nature of the Execution Function comes with the body of the Execution Function, so problematic parts of the code are clearly identified and labeled as such. This is for instance the case in C# where the keyword "`async`" can be used when defining a method. The downside is that it adapts poorly to all execution platforms: perhaps an Execution Function designated as "non-blocking" by the initial developer actually runs very fast on another machine, and thus could instead be executed in a blocking manner. The second solution is thus more adaptable, since the blocking/non-blocking nature is specified "later" in the process, *i.e.*, in the Communication Protocol for each Mapping. Moreover, we believe it makes more sense that the "caller" of the Execution Function (*i.e.*, in our case, the Communication Protocol) decides how it executes it ; rather than be forced to do it in a particular way, without any knowledge of it.

In Listing 3.5, we show the pseudo-code corresponding to the specification of the non-blocking nature of the Execution Function call of the Mapping "EvaluateGuard".

Listing 3.5: Specifying the non-blocking nature of a Mapping of the Communication Protocol of fUML, in pseudo-code.

```
1 Mapping EvaluateGuard:
2   upon evaluateGuard
3   triggers ActivityEdge.evaluateGuard() nonblocking
```

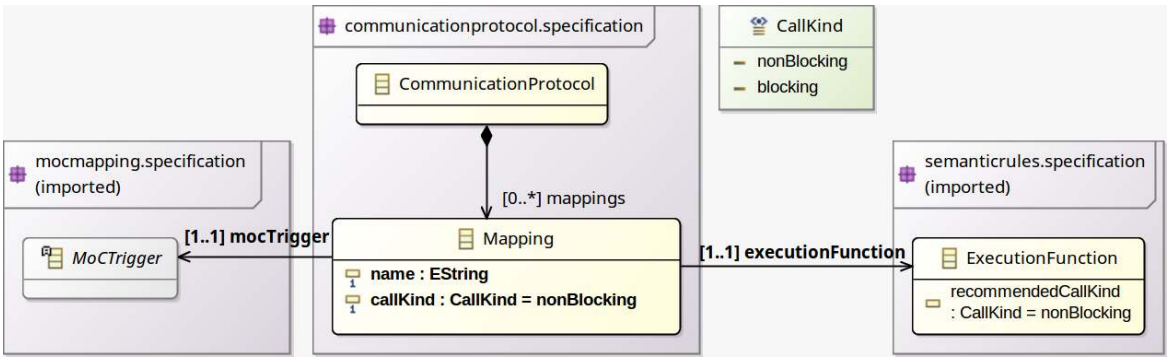

Figure 3.21: Excerpt from the metamodel of the Communication Protocol showing the blocking or non-blocking nature to an Execution Function can be specified.

This solution can be improved by adding the possibility to specify the blocking/non-blocking nature at the Execution Function level as well. In that case, the idea is to use it as a guidance (or default choice), *i.e.*, that if a Mapping does not explicitly specify the nature then it is looked up at the Execution Function specification. The editor of the Communication Protocol metalanguage can also leverage this information to suggest the nature to the Communication Protocol Designer. Ultimately, the decision remains in the hands of the Communication Protocol designer as we originally suggested.

Figure 3.21 shows an excerpt from the metamodel of the Communication Protocol extended with the feature we just described.

Runtime of Non-blocking Execution Function Calls

When the Execution Engine executes a MappingApplication specifying that the associated Execution Function call is non-blocking, then it launches it in a non-blocking manner. There are two ways this can happen: either the runtime of the Semantic Rules (the Executor) provides the means to execute an Execution Function call in a non-blocking manner, or it does not, in which case the runtime of the Communication Protocol is in charge of launching the execution in a non-blocking manner.

Implementing a non-blocking method call depends on the language (and its execution platform) used to implement the runtime of the Semantic Rules or of the Communication Protocol. For instance for JVM languages, the java.util.concurrent[3] package provides useful classes and methods to implement this. Python users may use the `asyncio` module[4], while in Ruby one may use Fibers [5].

[3]http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
[4]https://docs.python.org/3.4/library/asyncio.html
[5]http://ruby-doc.org/core-2.1.1/Fiber.html

Listening for Completion

When the Execution Engine has launched a non-blocking Execution Function call, it must keep a reference to the place where the result of the call will be stored. In many GPLs, this is implemented as a "Future" or "Promise". The Execution Engine must then check as often as possible whether or not the call has completed (*e.g.*, on the JVM by using the method `java.util.concurrent.Future.isDone()`), or use some form of signal (*e.g.*, through the Observer design pattern) to be notified of the completion of the call.

Managing Completions

When an Execution Function call has completed, it can have an influence on the next allowed steps of execution. The Execution Engine must thus re-compute the set of possible Scheduling Solutions upon completion of any Execution Function call.

Interruption of an Ongoing Execution Function Call

Finally, in case an ongoing call is blocked or should be interrupted because another part of the model took precedence, we propose to enable Mappings of the Communication Protocol to interrupt an ongoing Execution Function call. Listing 3.6 shows the pseudo-code of the specification of such a Mapping.

Listing 3.6: Specifying the Mapping to interrupt an ongoing non-blocking Execution Function call, in pseudo-code.

```
1 Mapping InterruptEvaluateGuard:
2   upon interruptEvaluateGuard
3   interrupts EvaluateGuard
```

At runtime, when a MappingApplication corresponding to this Mapping occurs, it interrupts the ongoing Execution Function call corresponding to the mapping "EvaluateGuard" if there is one, otherwise it does nothing. Depending on how the Semantic Rules metalanguage applies changes to the model being executed, it may not be possible to revert the side-effects that the partially-executed Execution Function call has made on the model. This feature can thus lead to an inconsistent state of the model. To palliate this, the changes to the model can be applied using transactions [134].

Likewise, continuing an interrupted Execution Function Call is possible, so long as that same Mapping has not be triggered in-between. Listing 3.7 shows the pseudo-code of the specification of such a Mapping.

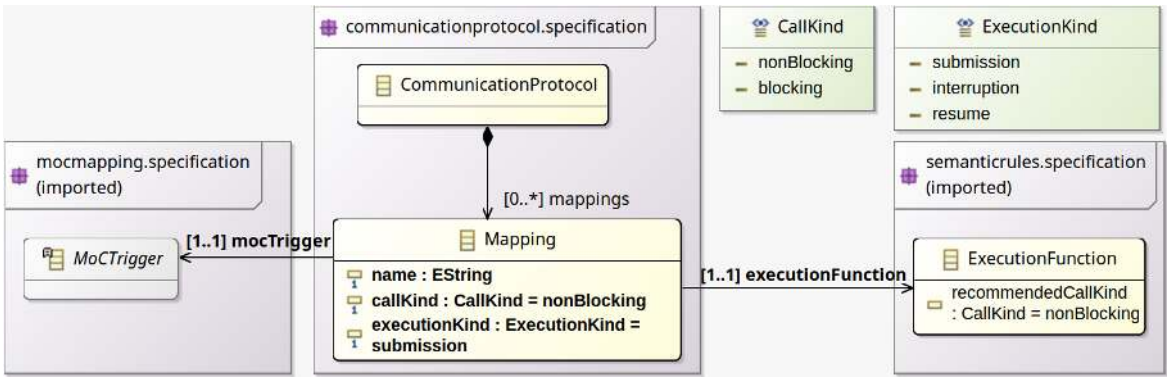

Figure 3.22: Excerpt from the metamodel of the Communication Protocol showing the different natures of calls to an Execution Function: a submission (to start executing it), an interruption (to halt an ongoing non-blocking call) or a resume (to start an interrupted call).

Listing 3.7: Specifying the Mapping to resume an interrupted non-blocking Execution Function call, in pseudo-code.

```
1 Mapping InterruptEvaluateGuard:
2   upon interruptEvaluateGuard
3   resumes EvaluateGuard
```

At runtime, such a MappingApplication resumes the previous ongoing non-blocking Execution Function Call corresponding to the mapping “EvaluateGuard” if there is one, otherwise it does nothing.

Figure 3.22 shows an excerpt from the metamodel of the Communication Protocol extended with these possibilities.

3.4.4 Costs and Downsides

The main cost of this feature is that it ties the execution of a model to the physical machine on which it is executed. Indeed, the real physical time taken by the machine to perform the non-blocking Execution Function call varies, and as such, the model may be in different states upon completion of the call, in different executions of the same model, possibly altering the rest of the execution. Previously, a model execution only relied on “logical time”, *i.e.*, the causalities between the MoCApplicationTriggers, agnostic of the physical machine used. The main downside is that it hinders the replayability of scenarios. Before, a scenario was essentially composed of the set of arbitrary choices realized by the heuristic

of the runtime. Now, the completion of non-blocking Execution Function calls must also be taken into account. This means that a model execution may be made smoother thanks to non-blocking calls, but this smoothness is not guaranteed on all possible platforms on which the execution is performed.

Finally, this feature can be technically difficult to implement because modifications to the model may be performed in a non-blocking Execution Function call, thus possibly making difficult the use of external tooling depending on the model's runtime state. For instance this is the case in our implementation of the graphical animation, which assumes a certain transaction protocol for the modification of the model (*i.e.*, EMF Transactions[6]). This protocol is disrupted by the modifications conducted in non-blocking calls. A solution for this, albeit requiring significant implementation effort, would be to use a metalanguage for the Semantic Rules that can automatically wrap model modifications to make use of the protocol used to modify the model.

3.4.5 Feature Summary

The seminal approach is fully sequential, in the sense that each execution step is executed and completed entirely before the next one starts. With this feature, the approach effectively becomes concurrent because non-blocking Execution Function calls are allowed to span over several execution steps. They must be designed carefully, so as to not provoke data races issues. But this feature is necessary for the support of xDSMLs whose behavior is rooted in the "real world", *i.e.*, when relying or controlling an external system like a robot or a sensor, whose execution lasts significantly more than regular Execution Function calls. It makes the simulation more user-friendly, since the end user does not always have to wait for previous execution steps to have finished executing (particularly in the case where it would have no impact on the future of the execution). It also enables some of the other features presented in this chapter.

3.5 Completion of an Execution Function Call

The completion of an Execution Function call is a meaningful event during a model's execution. It represents the end of the behavior of a model element, and is usually indicative of the next steps to perform. In this section, we only consider Execution Function calls which do not return any data (*i.e.*, their return type is Void). So far the completion of such calls is not represented explicitly in the concurrency model. In fact, for blocking Execution Func-

[6]http://www.eclipse.org/emf-transaction/

tion calls, they are confounded with the start of the Execution Function call, since they are seen as instantaneous. This is no longer the case with non-blocking Execution Function calls, and as such their beginning and completion should be treated as separate. In this section, we study the purpose of this separation, and how it can be implemented.

3.5.1 Purpose

With the introduction of non-blocking Execution Function calls in the previous section, comes the separation of the *beginning* of a call from its *completion* (or ending). So far, they were considered instantaneous and therefore represented in the concurrency model as a single event in the Event Structure. But for non-blocking calls, the completion most likely does not happen in the same execution steps as the beginning.

Let us consider two Events, $e\_retrieveData$ and $e\_displayData$. Our goal is to first "retrieve data", *e.g.*, evaluating a guard, retrieving the current temperature, etc., and then "display data", *e.g.*, by printing it to the standard output. The causality between these two events is initially denominated as $e\_retrieveData < e\_displayData$. For the sake of this section, let us now consider that retrieving the data takes some time, and therefore it should be performed in a non-blocking call so as to not block the rest of the execution (*i.e.*, typically the case if the data must be retrieved from an external component over a network, etc.). For the sake of generality, we consider that displaying the data also takes time and should be done in a non-blocking manner.

Both events should thus be captured in the concurrency model as a couple of event corresponding to their beginning and their completion. For $e\_retrieveData$, this means that we now have $e\_begin\_retrieveData$ and $e\_end\_retrieveData$ with the following causality:

$$e\_begin\_retrieveData < e\_end\_retrieveData$$

The same happens for $e\_displayData$ with the causality:

$$e\_begin\_displayData < e\_end\_displayData$$

The causality between retrieving and displaying the data can thus be specified as:

$$e\_end\_retrieveData < e\_begin\_displayData$$

3.5.2 Challenges

Compared to the initial description of the approach, there are two issues that must be dealt with:

- Event *e_*foo initially represents both *e_begin_*foo and *e_end_*foo. They must be made explicit with the causality *e_begin_*foo $<$ *e_end_*foo.
- Event *e_end_*foo is a bit peculiar because its occurrences represent something that happens in the Semantic Rules, *i.e.*, the completion of an Execution Function call. This means that it may only occur whenever, in the model, the corresponding Execution Function call has completed. Its occurrences are thus resulting from the runtime of the model, whereas occurrences of *e_begin_*foo *drive* the runtime of the model.

The former is just a matter of MoCMapping design. The latter is a bit more complex. We propose to name as *Controlled Events* (respectively *Controlled EventTypes*) the events from an Event Structure (respectively, the EventTypes from an EventType Structure) whose occurrences we plan to finely control based on other concerns of the semantics of the xDSML.

In our case, *e_end_*foo is a Controlled Event, subjected to the completion of the Execution Function call mapped to *e_begin_*foo. In the rest of this section, we show how such EventTypes are specified in the MoCMapping, and describe how they are controlled at runtime.

3.5.3 Specification

The EventType *et_*foo must be replaced by two EventTypes *et_begin_*foo and *et_end_*foo with a causality between them. There are several ways to accomplish that. The meta-language used for the MoCMapping can provide a language construct corresponding to this structure. It can also be simply considered as a design pattern to be used during the design of the MoCMapping. In our description, we choose the latter so as to keep our Event/EventType Structures "pure" (*i.e.*, unaltered by our approach).

Listing 3.8 shows an example specification, in the MoCMapping, of this design pattern, using pseudo-code.

Listing 3.8: Example specification of , specified using pseudo-code.

```
1 context MyData:
2   EventType et_begin_retrieveData;
3   EventType et_end_retrieveData;
```

```
4   EventType et_begin_displayData;
5   EventType et_end_displayData;
6
7   constraint beginRetrieveBeforeCompletion:
8     self.et_begin_retrieveData
9       strictly alternates self.et_end_retrieveData;
10
11  constraint beginDisplayBeforeCompletion:
12    self.et_begin_displayData
13      strictly alternates self.et_end_displayData;
14
15   constraint retrieveBeforeDisplay:
16     self.et_end_retrieveData
17       strictly alternates self.et_begin_displayData;
```

In this example, the constraint `strictly alternates` between two Events e_foo and e_bar means that the i^{th} occurrence of e_foo happens strictly before the i^{th} occurrence of e_bar, which happens strictly before the $(i+1)^{th}$ occurrence of e_foo. This ensures that for every beginning there is always a corresponding completion, and that for every retrieval of data there is always a corresponding displaying of it. This is formalized as follows:

$$\forall i \in \mathbb{N}, e\_foo_i < e\_bar_i < e\_foo_{i+1}$$

In the previous section, we have presented non-blocking Execution Function calls and the possibility to interrupt them. Considering the constraints we have defined, interruption is not possible for the example we have just given. Additional EventTypes and constraints should be specified to enable the use of the interruption mechanism we have defined.

In order to finely control the Controlled EventTypes, we must specify to which Execution Function call they correspond. This is done in the Communication Protocol, as illustrated on the pseudo-code specification shown on Listing 3.9.

Listing 3.9: Pseudo-code specification of a Mapping whose Execution Function completion is represented explicitly in the concurrency model

```
1 Mapping RetrieveData:
2   upon et_begin_retrieveData
3   triggers MyData.retrieveLatest() nonblocking
4   raises et_end_retrieveData
```

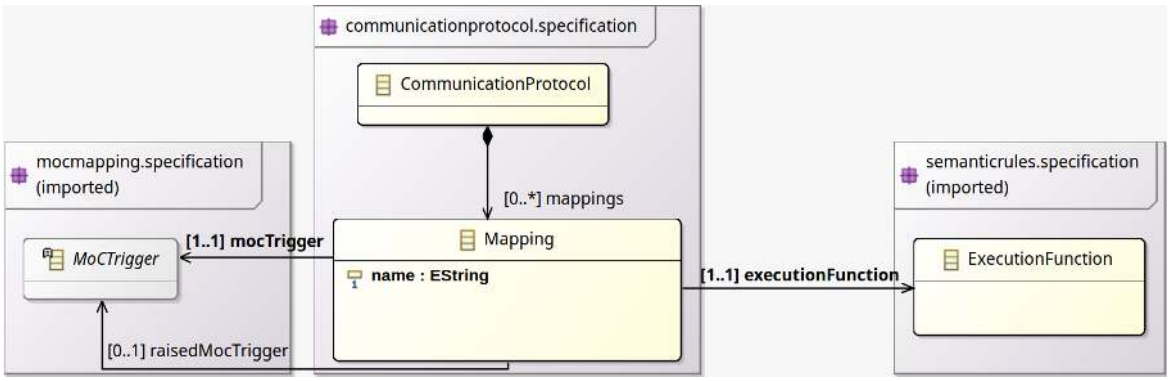

Figure 3.23: Excerpt from the metamodel of the Communication Protocol showing the possibility for a Mapping to raise a MoCTrigger as a marker of the completion of its Execution Function.

In this listing, the Mapping "RetrieveData" raises the MoCTrigger (EventType) *et_end_retrieveData* when its corresponding Execution Function, "MyData.retrieveLatest()" has finished its execution.

Figure 3.23 shows an excerpt from the metamodel of the Communication Protocol extended with this possibility.

3.5.4 Runtime

Managing the Controlled Events could be realized partly in the MoCMapping side of things. However, to avoid relying on a particular implementation technology of the concurrency aspects, we describe the runtime as a layer added to the runtime of the Communication Protocol, thus making it compatible with any implementation of the MoCMapping.

The runtime for these events is as follows. First, they are identified by the Execution Engine by going through the Communication Protocol Application and gathering all the events specified in the "raises..." clauses. Since these Events must be controlled finely, they are filtered at every step by the Execution Engine. This means that, at every step, the default behavior of the engine is to filter out the Scheduling Solutions with occurrences of the Controlled Events.

This filtering out is disabled, for a Controlled Event, temporarily upon completion of an Execution Function call whose "raises..." clause is that Controlled Event. It is disabled until a solution with an occurrence of the Controlled Event has been selected by the heuristic of the engine. This ensures that, upon completion of an Execution Function call, one occurrence of the raised event has happened. This, in turn, guarantees that the MoCApplication accurately depicts what has effectively happened in the runtime of the Semantic

Rules. Figure 3.24 shows the updated sequence diagram of one step of execution (*cf.* Subsection 3.2.5). Between execution steps, the engine listens for the completion of ongoing Execution Function calls, thus impacting the behavior of method `removeSolutionsWithUnallowedControlledEvents` because the corresponding raised events are temporarily allowed.

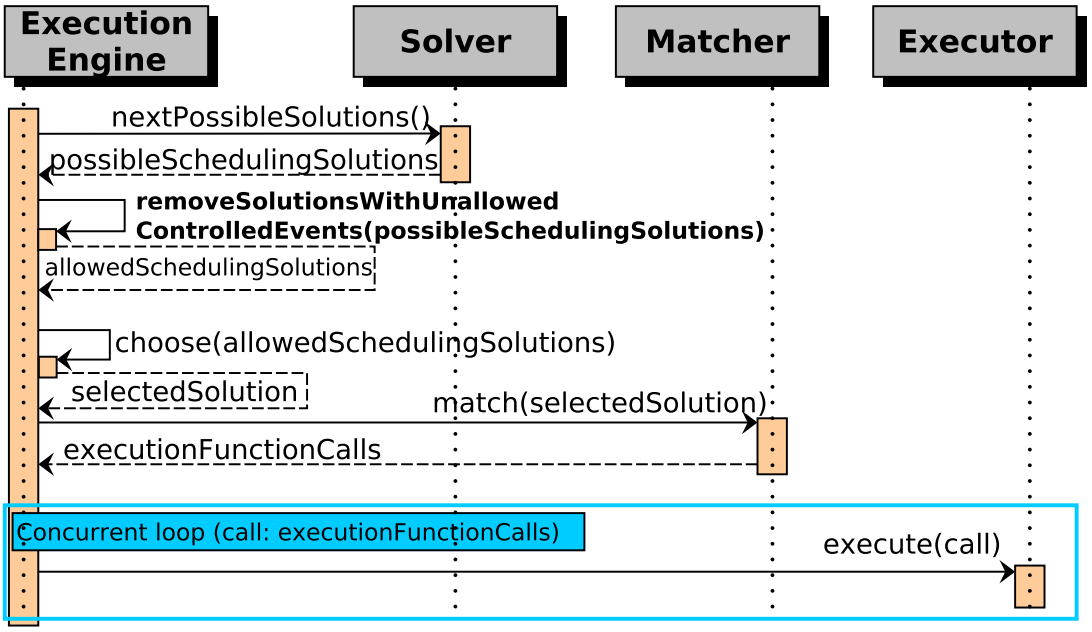

Figure 3.24: Sequence Diagram of a step of execution, with fine control of the Controlled Events.

3.5.5 Compatibility with Blocking Execution Function Calls

The issue we have described is mainly the consequence of the non-blocking Execution Functions call feature we have described in Section 3.4. Still, the solution presented above remains compatible with blocking calls. Compared to what we have presented, the constraint used must be loosened a little bit by removing the "strict" aspect. Listing 3.10 shows the adapted example MoCMapping specification in pseudo-code.

Listing 3.10: Excerpt from the MoCMapping specifiation, using pseudo-code, illustrating the causality relation between the "begin" and the "end" event when using blocking Execution Function calls.

```
1 context MyData:
2  constraint beginRetrieveBeforeCompletion:
3   self.et_begin_retrieveData
4    alternates self.et_end_retrieveData;
```

The constraint "alternates" between two Events e_foo and e_bar is such that:

$$\forall i \in \mathbb{N}, e\_foo_i <= e\_bar_i <= e\_foo_{i+1}$$

In other words, the i^{th} occurrence of e_foo happens before (possibly at the same time) the i^{th} occurrence of e_bar, which happens before (possibly at the same time) the $(i+1)^{th}$ occurrence of e_foo. However, two occurrences of the same event cannot occur at the same time, so e_foo_i and e_foo_{i+1} cannot occur simultaneously.

Moreover, there should not be any additional constraint preventing the simultaneous occurrence of the "begin" and "completion" events.

The only change in the runtime is that the engine should make sure that if a Mapping-Application specifies a blocking Execution Function call, then the selected Scheduling Solution must also contain an occurrence of the corresponding completion Event.

3.5.6 Feature Summary

This feature essentially implements an encoding of an asynchronous execution into the Event Structure formalism, through a backward communication (*i.e.*, from the Semantic Rules to the MoCMapping). All Execution Function calls can be encoded this way since synchronous executions can be seen as a particular case of asynchronous executions (*i.e.*, where the "begin" and the "end" events occur simultaneously). However, concurrency-aware xDSMLs are generally designed for simulations, rather than for implementations of real-world systems. This means that non-blocking calls are the exception rather than the norm, which is why this feature is presented as an "opt-in" option rather than an "opt-out" one.

3.6 Data-dependent Language Constructs

The semantics of some language constructs features a data-dependent control flow: that is, a control flow which depends on data available at runtime in the model. This is for instance the case of conditionals, for which the evaluation of a condition expression determines the next instructions to execute. In the concurrency-aware approach so far, the control flow of language constructs, captured in the MoCMapping, does not allow for runtime data to influence the future of the execution. We first motivate the importance of such language constructs, then identify the mechanism to enable their specification, and propose a solution that integrates into the concurrency-aware approach presented so far.

3.6.1 Purpose

To understand the prevalence of these language constructs, we consider the classification of control flow constructs in workflow systems, proposed in [133] and mentioned in Section 2.1. In this study, the authors have identified 43 patterns describing the control flow perspective of workflow systems (defined using formalisms such as BPMN [158], UML Activity Diagrams [111], BPEL [120], etc.). Among these patterns, 9 have semantics which, described using our approach, would rely on changing the control flow according to data available at runtime in the model. Patterns depending on the evaluation of a condition expression (*e.g.*, *Exclusive Choice*, akin to fUML DecisionNode; *Multi-Choice*, akin to a UML ForkNode with guards on outgoing branches; etc.) are typically concerned. Patterns based on iterations (*e.g.*, *Arbitrary Cycles*, corresponding to loops based on *goto* statements; *Structured Loop*, corresponding to repetitions based on dedicated language constructs such as *while...do* or *repeat...until*) also rely on the evaluation of a condition expression. As stated by the authors, "*Although initially focused on workflow systems, it soon became clear that the patterns were applicable in a much broader sense*" and "*Amongst some vendors, the extent of patterns support soon became a basis for product differentiation and promotion.*" [133].

We argue that, considering the number of patterns involved, this shows that many language constructs are concerned by this issue. Not being able to specify them entirely in concurrency-aware xDSMLs is thus problematic. In the rest of this section, we will study the mechanism required for the complete specification and execution of such language constructs, and propose a pragmatic solution for its integration into our approach.

3.6.2 Illustrative Example

In fUML, DecisionNodes represent decision points where one of the branches will be executed, based on the incoming data and the evaluation of the guards of the branches. In our example model, depending on the drink found, either "DrinkCoffee", "DrinkTea" or "DrinkWater" will be executed. Figure 3.25 shows a close-up of the simplified Event Structure for our example model, in the case where we found "Coffee" on the table. When the

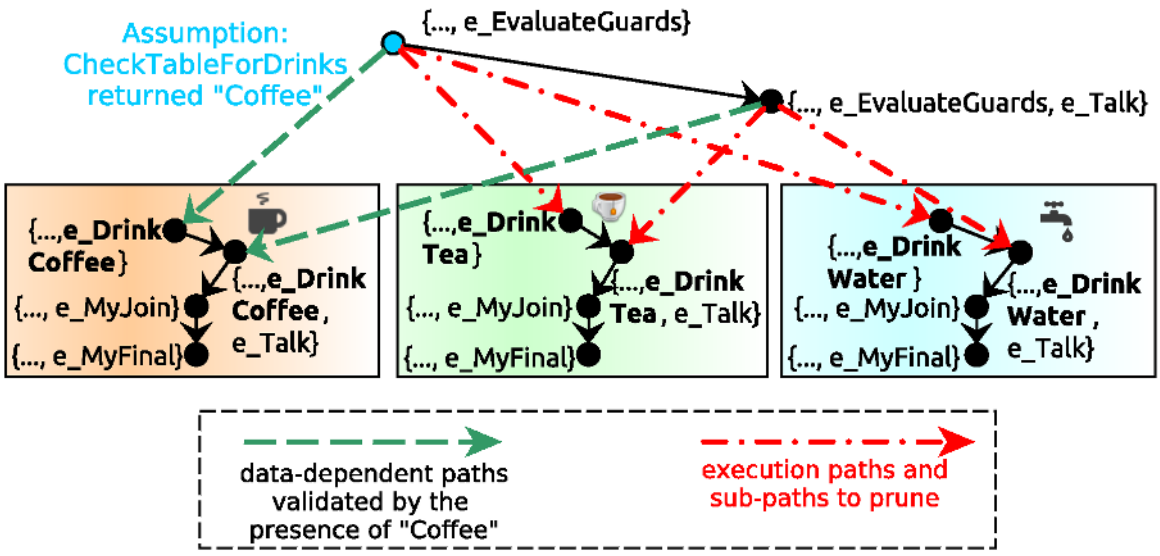

Figure 3.25: Close-up on the simplified Event Structure of the example Activity. We assume that earlier, the node "CheckTableForDrinks" returned "Coffee". Coloured lines represent the data-dependent causalities. The green dashed ones are the causalities validated by the presence of "Coffee". Red dots-and-dashes lines represent the execution paths that must be pruned because they are not consistent with the presence of "Coffee".

evaluation of the guards is realized, they return a boolean value defining whether or not their branch may be executed. In the Event Structure, all possibilities are represented. But since the concurrency model is data-independent, there is no connection between the result of the evaluation of the guards (*e.g.*, the boolean values returned by the Query `evaluateGuard()`) and the execution of one of the branches.

In this section, we study how to specify an interpretation of the result of a Query (*e.g.*, a boolean value in our case) so as to forbid some scenarios in the Event Structure (*e.g.*, by allowing and disallowing the branches of the ForkNode).

3.6.3 Challenges

The problem lies in a lack of communication between two concerns of the semantics. On the one hand, the concurrency model specifies, in a data-independent manner, all the possible execution scenarios. On the other hand, the Semantic Rules describe the dynamic data of the model and how they evolve at runtime. What is lacking is a means to specify, based on data available at runtime in the model, how to only consider the corresponding valid scenarios in the concurrency model.

The first step of the mechanism consists in retrieving from the model the piece of data which must be used to impact the control flow. For this, we rely on the taxonomy of Execution Functions we have presented in Section 3.3. When a Query is executed, it returns a piece of data from the model. So far, there is no way to exploit this data in order to only enable execution scenarios which are consistent with it. In the rest of this section, we will study how this can be specified and how it is implemented at runtime. The solution must maintain the modularity of our initial approach, particularly the separation of concerns and the data-independence of the concurrency model.

3.6.4 Extending the Communication Protocol

Since the communication we wish to specify links the Semantic Rules to the MoCMapping; the Communication Protocol, which already specifies a communication from the MoCMapping to the Semantic Rules, is an adequate candidate for the specification of this communication. We distinguish these two communications, and denominate them respectively *Mapping Protocol* (*i.e.*, the definition of Mappings as seen previously) and *Feedback Protocol* (*i.e.*, specification of how to interpret the values returned by queries to impact the control flow). The former is composed of *ModifierMappings* which map MoCTriggers (*i.e.*, EventTypes) from the MoCMapping and Modifiers from the Semantic Rules. The latter is constituted of *QueryMappings* mapping MoCTriggers and Queries, enhanced with an additional specification we call the *Feedback Policy*.

Figure 3.26 shows a Class Diagram of the approach with our changes.

In this context, the result of a Query is denominated as the *Feedback Value*. Its interpretation by the Feedback Policy will be used, at runtime, to only allow Scheduling Solutions (*i.e.*, in the case of Event Structures, configurations) consistent with the runtime state of the model.

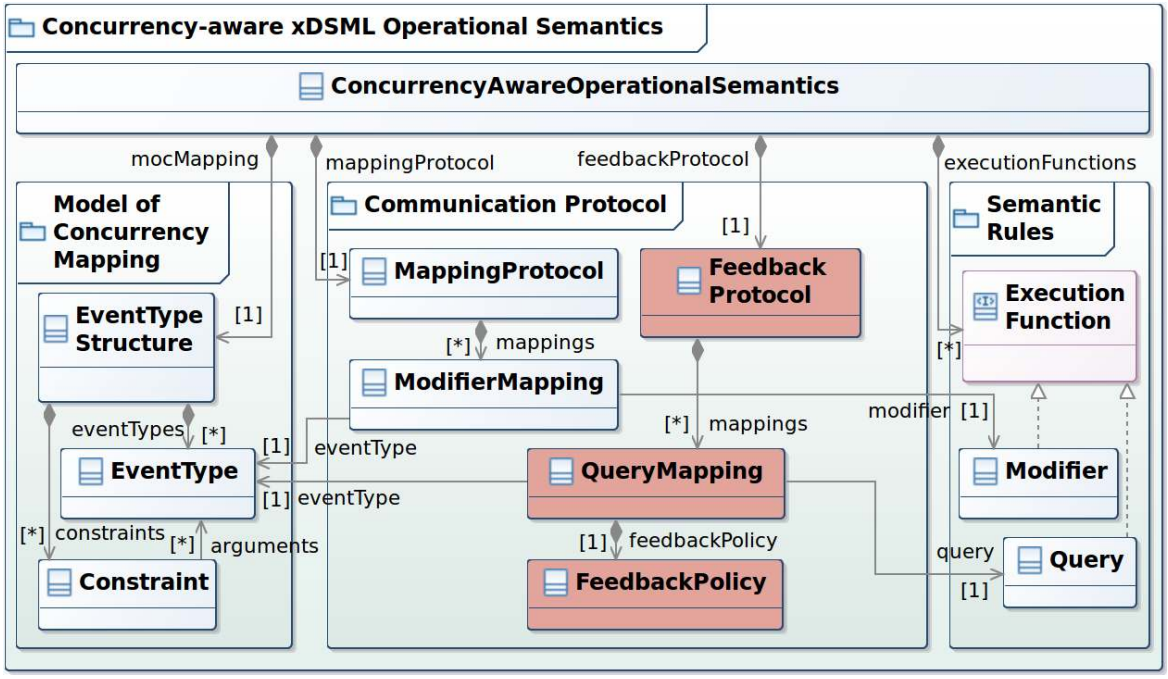

Figure 3.26: Metamodel of the approach including the separation of the Communication Protocol. New concepts related to the Feedback Protocol are in red. See Subsection 3.2.2 for the definition of the MoCMapping/EventType Structure, and Section 3.3 for the taxonomy of the Execution Function.

Feedback Policy

Let us detail the Feedback Protocol and how it is applied. We first consider an Event Structure E. E is formally defined by $< Evt, \mathbb{C}, \vdash >$, where Evt is a set of Events, $\mathbb{C}$ is an ordered set of consistent configurations and $\vdash$ is the enabling relation [160]. A configuration is a set of events that have occurred by some stage in the process. Also, any event in a configuration should have been enabled by another event in a previous configuration (or by the null set for uncontrolled events like the initial one). We denote $path(c_1, c_2)$ two "causal" configurations, *i.e.*, two configurations such that:

$$\exists e \in c_2, c_1 \vdash e \wedge \nexists c \in \mathbb{C}, c < c_1 \wedge c \vdash e$$

In other words, the configuration c_2 contains at least one event directly enabled by c_1.

Based on this, we can define an event structure as a triplet $< Evt, \mathbb{C}, \mathbb{P} >$ where $\mathbb{P}$ is the set of paths between the configurations in $\mathbb{C}$. There exists two different kinds of paths in $\mathbb{P}$, *i.e.*, $\mathbb{P} \triangleq \mathbb{P}_I \cup \mathbb{P}_D$. $\mathbb{P}_I$ are the paths independent from the runtime state of the model while $\mathbb{P}_D$ are the data-dependent ones. Let us denote $rts(c)$ the runtime state of the model

at the configuration c of the Event Structure. When $path(c_1, c_2) \in \mathbb{P}_D$, and its dependency is towards the runtime state of the model at a specific configuration c, we denote it as $path(c_1, c_2)_{rts(c)}$. This means that depending on the runtime state of the model at a certain point c of the Event Structure (where c precedes c_1 and c_2), going from c_1 to c_2 *may be possible*. Let us denote as $f_{path(c_1,c_2)_{rts(c)}}$ the function that determines if the path from c_1 to c_2 may be taken, depending on an interpretation of $rts(c)$. It returns a boolean value: either the path is allowed or it is not.

Figure 3.27 represents these concepts for a generic Event Structure. The data-dependent path from c_1 to c_2 is conditioned by some data available at runtime in the model at the point of execution of the configuration c.

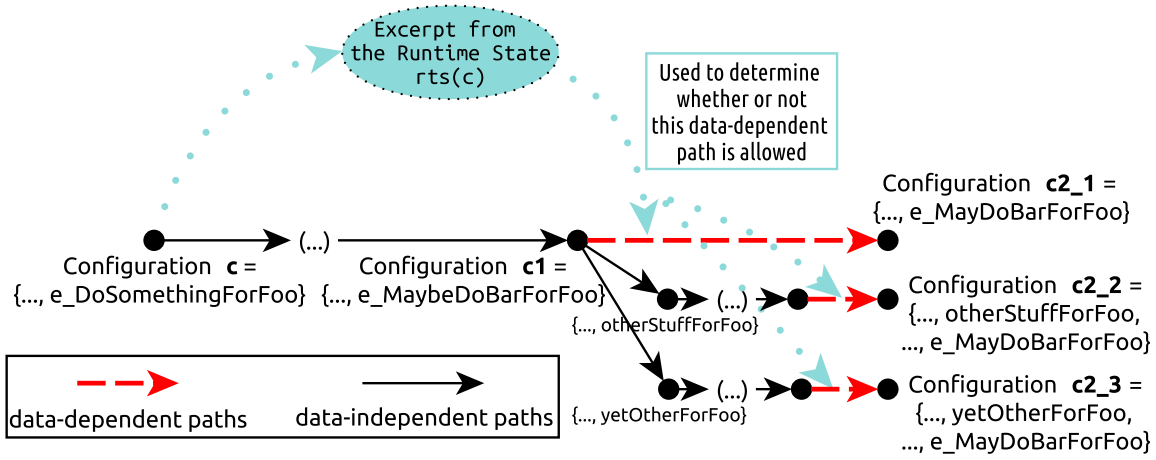

Figure 3.27: Illustration of the general principle of the Feedback Protocol.

The Feedback Protocol must specify (at the language level, *i.e.*, in intention) the set of data-dependent execution paths (*i.e.*, $\mathbb{P}_D$) together with the set of functions f_p (where f_p determines whether a path $p \in \mathbb{P}_D$ may be taken or not). This specification must be independent of any model, but be applicable to any model conforming to the abstract syntax of the language. For a specific model, applying the Feedback Protocol consists in removing the execution paths from $\mathbb{P}_D$ that are inconsistent with the runtime state of the model. It cannot add any paths in $\mathbb{P}$, nor remove any paths from $\mathbb{P}_I$.

Pragmatics of the Feedback Protocol

Practically, computing the whole Event Structure for a model may be complex or impossible. If the model is very large or highly parallelizable, then the exponential number of configurations and execution paths (possibly infinite) makes it either too costly to com-

pute or too big to be usable. Let us consider the minimal situation, where we are capable of computing only the children configurations of a configuration.

Since the event structure is only partially constructed during a specific execution of the model, we do not have access to all the paths (and furthermore not all the data-dependent paths). Therefore, applying the Feedback Protocol cannot consist in pruning execution paths in the event structure. Instead, the Feedback Policies only specify the EventTypes which are inconsistent with regards to the runtime state of the model, and at execution time *all the occurrences* of the corresponding instances of the EventType are forbidden. Forbidding an event from occurring results in pruning the corresponding execution path in the implicit event structure. However, it should not prune other occurrences of that same event which depend on another runtime state of the model.

To handle this issue, we add the following role to the Feedback Policy: its interpretation of the Feedback Value must return the set of EventTypes inconsistent with the runtime state of the model *and* the set of EventTypes which are data-dependent and consistent with the runtime state of the model. This way, the next occurrences of these consistent EventTypes are considered as the limit after which the occurrences of the inconsistent EventTypes do not represent a data-dependent decision anymore. Thus, after the consistent EventTypes have occurred, forbidding the inconsistent EventTypes ceases. This adds the following constraint: the concurrency model should not allow situations where different occurrences of the same event depend on Feedback Policies (possibly several occurrences of the same policy) which can be applied at the same time. When considering two queries, and their Feedback Policies overlap in terms of which events are compatible or incompatible, then the concurrency model should not allow these two queries to overlap the application of the Feedback Policy of the other query. This means that the second query should never be executed between an execution of the first query and occurrences of the compatible events of the Feedback Policy of that first query. More formally, the different data dependencies of the control flow for a model element must be treated sequentially (*i.e.*, no two dependencies on different pieces of data should intervene concurrently during the execution of the underlying Event Structure). Otherwise, it is possible that the MoCApplication falls in a state of deadlock, halting the execution. In the case of fUML, this means that when a guard is evaluated, it cannot be re-evaluated (because new tokens have arrived on the incoming edges) before the branch resulting of the first evaluation has started executing.

We show the pseudo-code specification of the Feedback Protocol for fUML on Listing 3.11. It relies on the specification of two additional EventTypes, declared in the context of ActivityEdges: $et\_mayExecuteTarget$ and $et\_mayNotExecuteTarget$. These Event-

Types are *Controlled EventTypes* as defined in Section 3.5. Their occurrences are managed by the rest of the runtime of the execution semantics of the xDSML.

Listing 3.11: Updated Feedback Protocol of fUML, specified using pseudo-code.

```
1 Mapping EvaluateGuard:
2   upon et_evaluateGuard
3   triggers ActivityEdge.evaluateGuard() returning result
4   feedback:
5     result = true => allow ActivityEdge.et_mayExecuteTarget
6     result = false => allow ActivityEdge.et_mayNotExecuteTarget
```

The following description is at the language level, but it is applied at runtime at the model-level.

1. Upon execution of the query `evaluateGuard()`, its boolean result is stored in the variable `result`.

2. If the result was true, then only the execution paths with an early occurrence of *et_mayExecuteTarget* are allowed. In other words, Scheduling Solutions with an occurrence of *et_mayNotExecuteTarget* are forbidden until a Scheduling Solution with an occurrence of *et_mayExecuteTarget* has been selected.

3. Otherwise (the result was false), then only the execution paths with an early occurrence of *et_mayNotExecuteTarget* are allowed. In other words, Scheduling Solutions with an occurrence of *et_mayExecuteTarget* are forbidden until a solution with an occurrence of *et_mayNotExecuteTarget* has been chosen.

Explicitly representing the case where a branch is not allowed (*i.e.*, via *et_mayNotExecuteTarget*) is required because otherwise, additional constraints of the MoCMapping cannot be defined. Indeed, in an Event Structure, it is not possible to reason over an Event not occurring. Since an Event Structure captures a partial ordering, there may be an indefinite number of steps between two occurrences. By capturing explicitly when a branch is not allowed, we enable the correct definition of the rest of the MoCMapping, *i.e.*, both cases are explicitly handled by the partial ordering.

Figure 3.28 illustrates how the Feedback Protocol intervenes in the case of the example fUML Activity. For representation purposes, only one of the guard is considered for this figure. In reality, the guards are concurrent and the other branch of the ForkNode is also concurrent so there are a lot of possible execution paths.

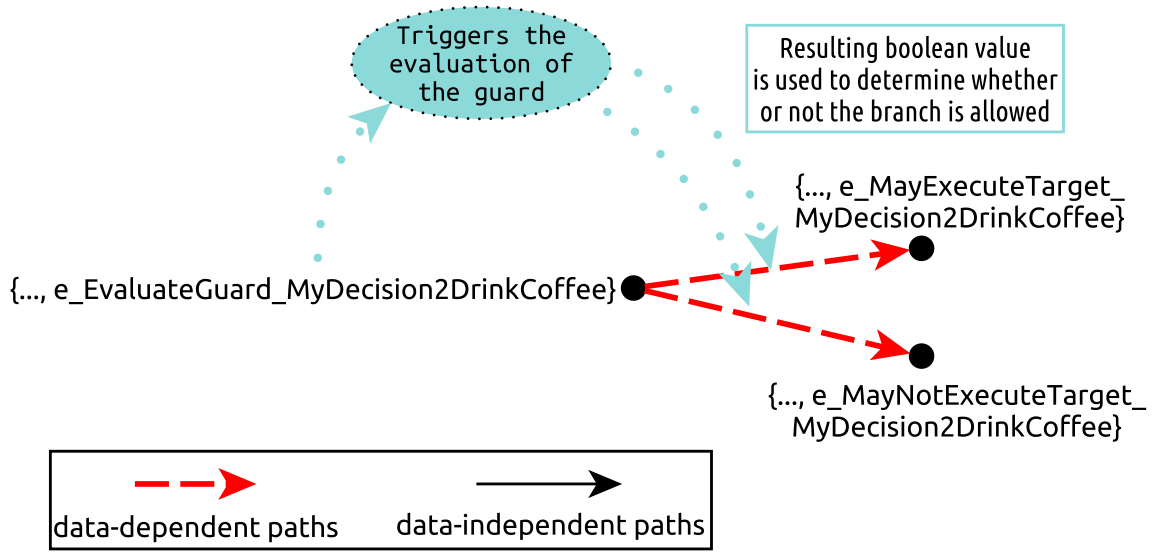

Figure 3.28: Closeup on the part of the Event Structure of the example fUML Activity in which the Feedback Protocol is used.

The target EventTypes (*i.e.*, $et\_mayExecuteTarget$ and $et\_mayNotExecuteTarget$ in our example) are Controlled EventTypes which means that at every step, the Scheduling Solutions containing their occurrences are filtered out by the execution engine, because they are only supposed to occur as a result of something happening in other parts of the semantics. In our case, they can only occur depending on the application of the Feedback Policy after a Query has returned a particular value.

An important aspect of the Feedback Protocol is that the metalanguages for the Feedback Policy and for the Semantic Rules must be able to communicate. More precisely, the value returned by a Query must fit within the type system used by the metalanguage for the Feedback Policy. In the pseudocode example shown above, the expression `result = true` is valid only if the expressions defined by the Feedback Policy metalanguage are compatible with the values returned by the Query `evaluateGuard()`.

Compatibility with non-blocking calls

We have described in Section 3.4.3 a feature to execute Execution Function calls in a non-blocking manner. It may be necessary to use it for Queries which may take a long time to compute, for instance if the data it returns is based on a complex computation. However, this requires some additional modifications of the runtime to ensure that the Feedback Protocol can be applied correctly.

First of all, the application of the Feedback Protocol results in some Scheduling Solutions being temporarily forbidden. Since a non-blocking call to a Query can complete at any time, the possible Scheduling Solutions presented by the heuristic of the execution engine should be updated as soon as possible after the completion of a Query call. This ensures that the choices made by the heuristic are realized based on the latest runtime state of the model.

Another modification to the runtime is to ensure that some arbitrary decisions cannot be made too early. The EventTypes targeted by the Feedback Policy are Controlled (*cf.* Section 3.5), which means that their occurrences are finely controlled by the runtime based on additional information from the semantics. In particular, after the Query has been launched, and before it has returned, no decision can be taken about them. This means that all Scheduling Solutions containing occurrences of the EventTypes targeted by the Feedback Policy associated to a Query which has been launched must be filtered out until the Query has completed its execution. This ensure that no early decision can be made about these EventTypes before the Feedback Data that conditions their occurrences has been retrieved.

3.6.5 Feature Summary

With this feature, we establish another backward communication from the Semantic Rules to the MoCMapping (the first one being in Subsection 3.5). This communication effectively improves the expressive power of the concurrency-aware xDSML approach, since data-dependent language constructs could not be handled correctly previously. This type of construct is core to many xDSMLs, as it is used for conditionals, switches, etc., which explains why the changes to the metalanguages of the approach are more voluminous than for previous features.

3.7 Composite Execution Functions

Execution Functions are initially designed with the intent of being called by the Executor (runtime of the Semantic Rules) under the impulsion of the Execution Engine. This naturally hinders the definition of *Composite Execution Functions*, which make use of other Execution Functions. This section is dedicated to enabling the definition of Composite Execution Functions while maintaining the concurrency-awareness of the language.

3.7.1 Purpose

Different language constructs must often communicate during their execution to realize their semantics. In our approach, this is concretized by the definition of a data exchanges between Execution Functions, that is, the output of some Execution Function is used as input for some other Execution Function. This includes situations where an Execution Function, during its execution, calls another Execution Function. In that case, the latter may even not return any data, in which case it is simply a means to reuse code (*i.e.*, a set of instructions bundled together as a function, procedure, subroutine, etc.).

Since Execution Functions are triggered by the Executor because a corresponding Mapping-Application has been matched on the selected Scheduling Solution (from the Solver), the possibility of defining data exchanges between Execution Functions is lost, the concurrency model being independent from the data concerns.

Data exchanges may still be defined, albeit using some form of pattern. For code reuse (*i.e.*, function call not returning any data), a common operation can be defined, exploited by both Execution Functions. This however requires a particular development methodology from the language designer (*i.e.*, it is not idiomatic to the Semantic Rules metalanguage). Listing 3.12 shows an example of this situation using pseudo-code. In this example, the Execution Function “caller” reuses another Execution Function “callee”. However, doing this directly hides away, from the concurrency model, the relation between “caller” and “callee” (the concurrency-awareness is lost). Listing 3.13 shows the solution to keep the concurrency-awareness, at the cost of being non-idiomatic for the Semantic Rules metalanguage.

For data exchanges, a piece of data can be stored in an attribute of a model element that is accessible to both Execution Functions, but this relies on an implicit protocol (*i.e.*, side effects on a common model element) and is also not idiomatic. Listing 3.14 shows an example of this situation using pseudo-code. In this example, the Execution Function “caller” calls another Execution Function “callee” with some argument “x”. However, doing this directly hides away, from the concurrency model, the relation between “caller” and “callee” (the concurrency-awareness is lost). Listing 3.15 shows the solution to keep the concurrency-awareness, at the cost of being non-idiomatic for the Semantic Rules metalanguage (and also requires the ability to call other Execution Functions, as presented in the previous example).

Listing 3.12: Example of an Execution Function which relies on the execution of another Execution Function.

```
1 public void caller(){
2   // ...
3   callee();
4   // ...
5 }
6
7 public void callee(){
8   // ...
9 }
```

Listing 3.13: Adaptation of Listing 3.12 so that the concurrency-awareness is preserved.

```
1  public void caller(){
2    // ...
3    commonOperation();
4    // ...
5  }
6
7  public void callee(){
8    commonOperation();
9  }
10
11 private void commonOperation(){
12   // ...
13 }
```

Listing 3.14: Example of a data exchange between two Execution Functions.

```
1 public void caller(){
2   // ...
3   var x = ...;
4   callee(x);
5 }
6
7 public void callee(T arg){
8   // ...
9 }
```

Listing 3.15: Adaptation of Listing 3.12 so that the concurrency-awareness is preserved. Requires additional adaptation to realize the call to "callee" as illustrated previously.

```
1  public void caller(){
2    // ...
3    var x = ...
4    self.foo = x;
5    callee();
6  }
7
8  public void callee(){
9    var arg = self.foo
10 }
```

Both cases require the adaptation of the Semantic Rules metalanguage. Since it allows the definition of the Execution Functions, it most likely also includes the possibility to call them (as is the case for most programming languages proposing the notion of operation, procedure, method, subroutine, etc.). We will refer to this feature as "*method-call*". This feature, in its common definition, does not combine well with our approach: Execution Function calls are supposed to be triggered by the Executor and not by other Execution Functions. We will propose a solution to make it compatible with our concurrency-aware approach, thus enabling the definition of what we call Composite Execution Functions.

3.7.2 Illustrative Example

In fUML, `Action` nodes may have OutputPins, *i.e.*, pins that deliver values to other nodes through object flows. In that case, an OutputPin is always executed after its owning node. Considering the concurrency-aware approach, there are initially two ways to specify this. First, we can wrap the execution of the OutputPin inside the execution of the owning node. This means that the execution of the pin is implicit, *i.e.*, not visible in the concurrency model. The second way consists in making it explicit in the concurrency model. The executions of the node and of the pin are thus separated, mapped by different MoCTriggers. One to represent the execution of the node, and one to represent the execution of the pin, with a causality between both to ensure that the latter is executed after the former.

In this section, we try to consider a mix of both solutions. The execution of the pin is considered as part of the execution of the node, facilitating the specification of the corresponding Execution Function, and possibly, the data exchange between the execution of a node and of its pin. But it is also scheduled explicitly in the concurrency model, *i.e.*, there is a causality between two Events, one representing the execution of the node and one representing the execution of the pin.

Let us illustrate the issue on a part of the example Activity (*cf.* Figure 3.2). Figure 3.29 shows the `Action` node "CheckTableForDrinks" and its OutputPin.

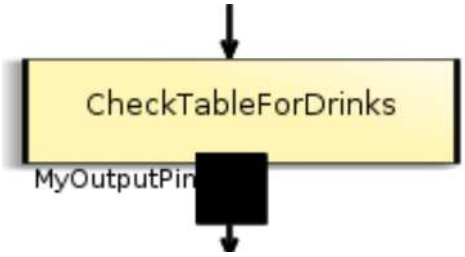

Figure 3.29: The `Action` node "CheckTableForDrinks" and its `OutputPin` from the example fUML Activity of Figure 3.2.

In the first solution, the execution of the pin is wrapped inside the execution of the node, so the corresponding Event Structure has only one Event, corresponding to the execution of the "CheckTableForDrinks" node (and implicitly of its pin). Figure 3.30 shows the overview of the different concerns in this situation.

In the second solution, the execution of the pin is separated from the execution of its owning node. For our example, this means that there are two different Execution Function calls, mapped through two different MappingApplications by two different Events of the

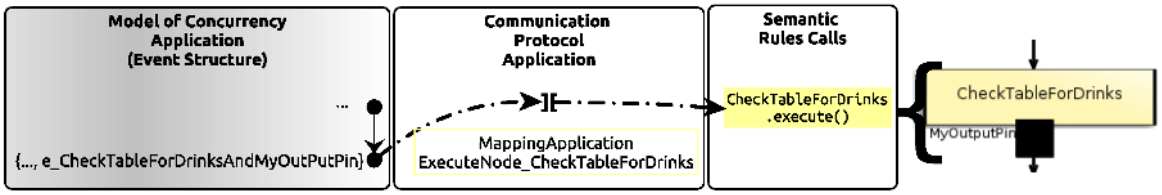

Figure 3.30: Overview of the semantics concerns for an excerpt of the example fUML Activity of Figure 3.2. The Execution Function for the Action node "CheckTableForDrinks" includes the execution of its pin "MyOutputPin".

Event Structure. Figure 3.31 shows the overview of the different concerns in this situation.

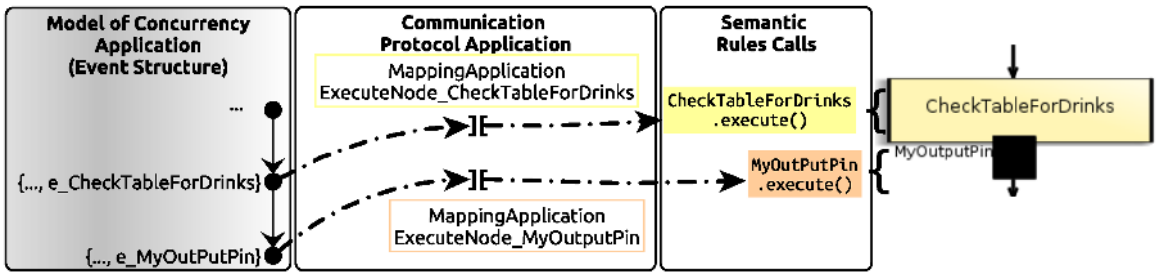

Figure 3.31: Overview of the semantics concerns for an excerpt of the example fUML Activity of Figure 3.2. The Execution Functions for an Action node and its pins are separated.

Our goal in this section is to be able to build the Event Structure of the second solution (which makes explicit the relation between the execution of "CheckTableForDrinks" and the execution of "MyOutputPin") while allowing the Execution Function Calls of the first solution (which gathers both executions into one Execution Function).

3.7.3 Challenges

More generally, let us consider two Execution Functions ef_{callee} and ef_{caller}. In the implementation of ef_{caller}, a method-call to ef_{callee} is realized. Our goal is to make sure the relation between ef_{caller} and ef_{callee} is captured accurately in the concurrency model, and to make sure that the runtime is consistent with the specification.

This feature requires a coordination between the MoCMapping and Semantic Rules specifications, and also between their respective runtimes. Indeed, since the execution of ef_{callee} is wrapped inside the execution of ef_{caller}, the representation of ef_{caller} in the concurrency model must be separated in two (*i.e.*, to differentiate the beginning of ef_{caller} from its end), similar to what was done in Section 3.4.

- **Challenge #1:** The concurrency must thus ensure the following relation: $launch(\text{ef}_{caller}) < launch(\text{ef}_{callee}) < return(\text{ef}_{callee}) < return(\text{ef}_{caller})$
- **Challenge #2:** The implementation of ef_{caller} should ideally not require specific code to implement the method-call to ef_{callee}. This means that we want to find a way such that, for the metalanguage of the Semantic Rules, method-calls to other Execution Functions are dealt with in a particular way (*i.e.*, enabling its "concurrency-awareness"), without having to change the syntax used for regular method-calls.

3.7.4 Solution

Our solution consists in adapting the specification of the concurrency concerns, the specification of ef_{caller} and the runtime.

Specification of the Concurrency Concerns

The consequence of this feature on the specification of the MoCMapping is that since the execution of ef_{callee} is contained by the execution of ef_{caller}, it must be captured as such in the concurrency model. Therefore, similar to the approach proposed for non-blocking Execution Functions in Section 3.4, a design pattern (or an equivalent language construct, depending on the MoCMapping metalanguage) can be applied to tackle Challenge #1 (*cf.* page 103) Figure 3.32 shows the corresponding Event Structure for a part of the example fUML Activity. The Events $e\_end\_MyOutputPin$ and $e\_end\_CheckTableForDrinks$ are Controlled Events, finely managed depending on the current state of execution of their corresponding Execution Function calls, as defined in Section 3.5.

Specification of the "Caller" Execution Function

There are two main constraints to the design of ef_{caller}.

First, it must be called in a non-blocking manner. Indeed, since the call of ef_{callee} by ef_{caller} is made explicit in the concurrency model, it will be triggered by an execution step of the Execution Engine. This execution step necessarily happens strictly after the step which initially triggered ef_{caller}. There may actually be any number of execution steps between the beginning of execution of ef_{caller} and the call to ef_{callee} by ef_{caller}. Executing ef_{caller} in a non-blocking manner allows the engine to first start the execution of ef_{caller}, while executing other steps until the execution of ef_{callee} happens.

The second constraint is that ef_{caller} must specify that it uses ef_{callee}. Indeed, the MoCMapping merely ensures that $launch(\text{ef}_{caller}) < launch(\text{ef}_{callee})$. But at runtime, we

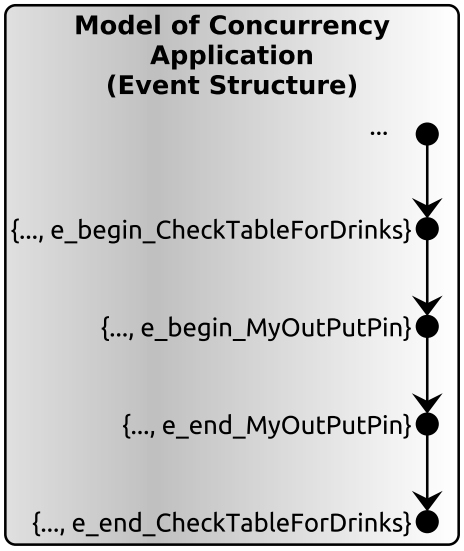

Figure 3.32: Event Structure representing the nested call of ef_{callee} by ef_{caller} for the excerpt of the example fUML Activity.

need to make sure that the execution of ef_{caller} has reached the point where it requires the execution of ef_{callee} before allowing a step triggering the execution of ef_{callee}. Therefore, in the example given previously, $e\_begin\_MyOutputPin$ is also a Controlled Event.

To do that, the Execution Engine needs to know which Execution Functions are called by which other Execution Functions, so that the corresponding MoCTriggers in the MoCMapping can be managed as Controlled EventTypes (*cf.* Section 3.4, their occurrences are finely controlled based on the rest of the semantics – in that case, on the current state of execution of ef_{caller}). This information can be specified in intention, or possibly inferred via static analysis of the Semantic Rules implementation.

Figure 3.33 shows an excerpt from the metamodel of the Semantic Rules illustrating the structure of Composite Execution Functions as we have described them. The main changes for this feature are located in the static semantics (ef_{caller} must be called in a non-blocking manner), and most importantly, in the runtime described below.

Changes to the runtime

Our solution is illustrated on Figure 3.34, which shows the modified Sequence Diagram for the Execution Engine.

It relies on modifying method-calls to Execution Functions. It works as follows. By default, the engine retrieves the declared callee Execution Functions and filters their oc-

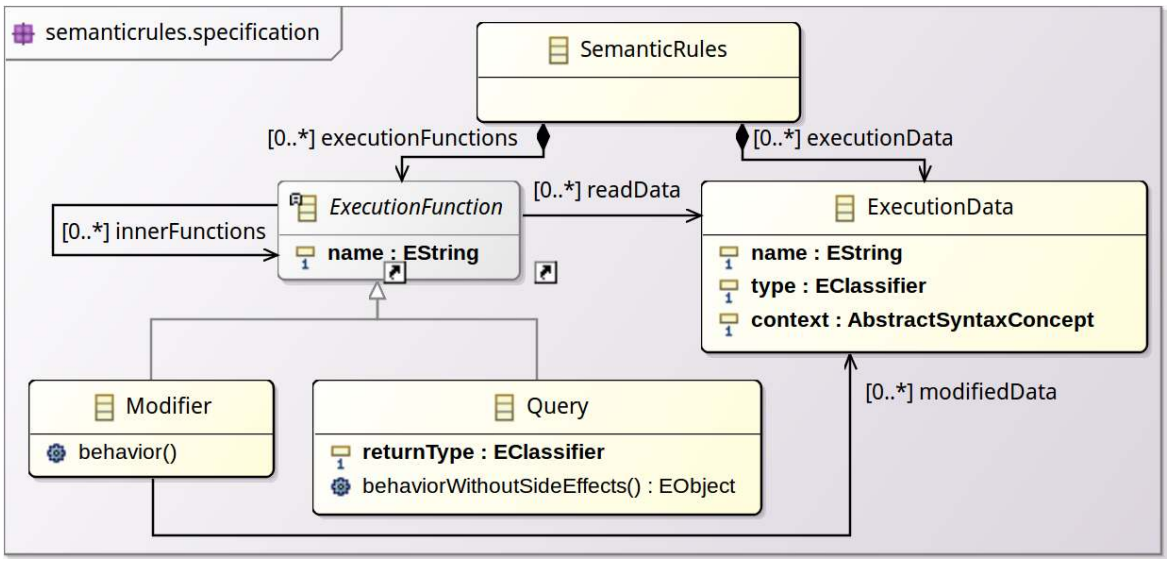

Figure 3.33: Excerpt from the metamodel of the Semantic Rules showing the structure of Composite Execution Functions.

currences out. This means that Scheduling Solutions leading to their execution are filtered out from the execution steps that can be selected by the heuristic of the engine.

When ef_{caller} is executed, instead of directly executing ef_{callee}, it sends a request to the Executor, which transmits it to the Execution Engine. This request contains the information that ef_{caller} is trying to execute ef_{callee}. Upon reception of this request, the engine disables the above-mentioned filter, so that an execution step leading to the execution of ef_{callee} may be selected. Meanwhile, ef_{caller} is put on hold, waiting for ef_{callee} being executed.

Once such a step has been selected and executed, and that ef_{callee} has completed its execution, the engine notifies ef_{caller}, which may proceed with the rest of its execution.

Modifications to the runtime include the communication between an Execution Function call and the Executor, between the Executor and the Engine, and then back from the Engine to the Executor and to the blocked Execution Function call. Additionally, the engine must also be able to filter out the solutions leading to the execution of ef_{callee} since its corresponding EventType is controlled.

This has the following drawback: there cannot be an occurrence of another Mapping launching the ef_{callee} Execution Function independently of what is happening in ef_{caller}. Indeed, there is no way to distinguish, in that case, an independent call to ef_{callee} from the call required by ef_{caller}. This is due to how, in the concurrency model, the abstract actions are represent independently of data from the model. This means that any call

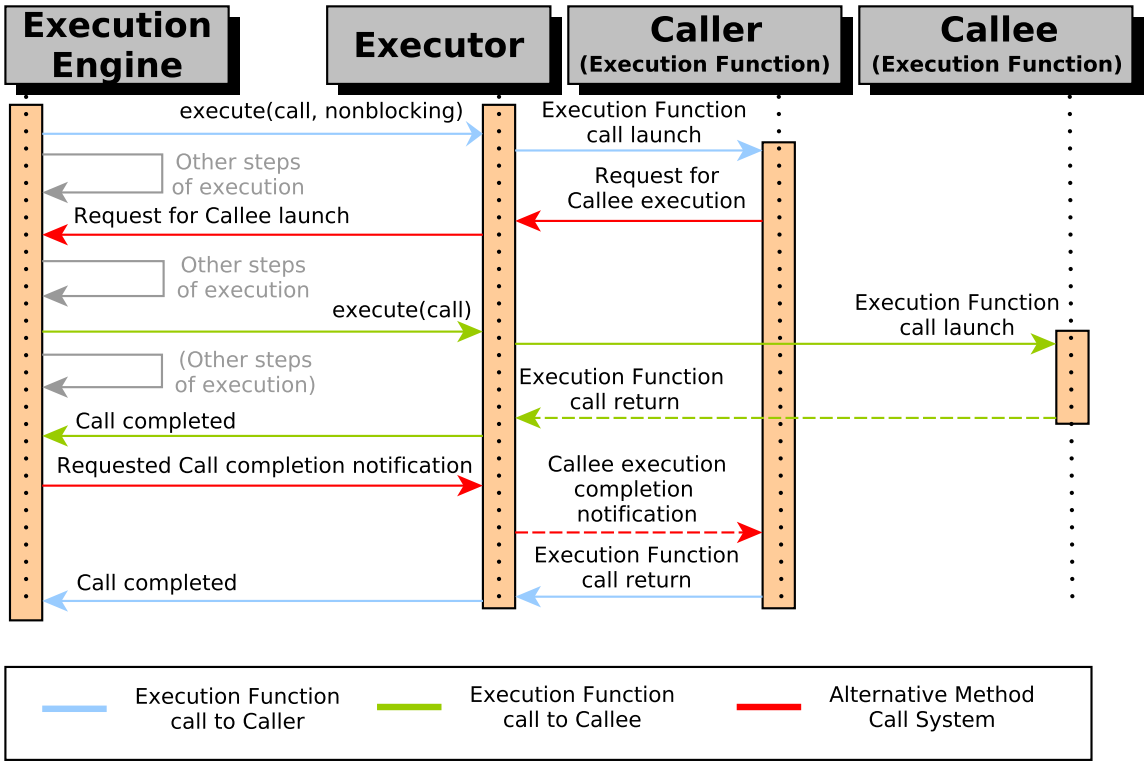

Figure 3.34: Modified Sequence Diagram of the Execution Engine to illustrate the reuse of an Execution Function ("Callee") by another Execution Function ("Caller") while making it explicit in the concurrency model.

to an Execution Function is considered independently from its calling context (*i.e.*, the runtime state of the model). This ultimately stems from the necessity to abstract away, in the concurrency concerns, parts of the model to enable its analysis and refinement. Therefore, to minimize the situations where an inconsistent state of the runtimes could be reached, the semantics of the xDSML should not include an independent call to ef_{callee} during the execution of ef_{caller}.

Implementation of the "Caller" Execution Function

Challenge #2 (*cf.* page 103) is restricted to the implementation of ef_{caller}. Ideally, the method-call mechanism should rely on the usual syntax, and be differentiated only at runtime depending on whether the operation called is another Execution Function or not.

There are several ways to divert the function call in the metalanguage for the Semantic Rules, more or less disruptive for the user code in the implementation of the Execution

Functions. Ultimately, the change should be equivalent to what is shown on Listings 3.16 and 3.17.

Listing 3.16: Example of user code implementing Execution Functions “caller” and “callee”.

```
1 public void caller(){
2   // ...
3   callee();
4   // ...
5 }
6
7 public void callee(){
8   // ...
9 }
```

Listing 3.17: Code actually executed corresponding to the specification shown on Listing 3.16.

```
1 public void caller(){
2   // ...
3   Executor.
     requestExecutionAndBlockFor(
     callee, []);
4   // ...
5 }
6
7 public void callee(){
8   // ...
9 }
```

One solution is to modify how the Semantic Rules Calls are generated based on a model. The modification consists in identifying the function calls which are made explicit in the concurrency model, and in transforming these calls according to the solution we have proposed. That is, to transform a call to ef_{callee} into a call request to the Executor and then blocking until notification that the call has indeed been realized. This solution is disruptive for the compiler of the Semantic Rules metalanguage but does not interfere with the user code.

3.7.5 Feature Summary

This feature relies on a complex back-and-forth communication between the Semantic Rules and the MoCMapping. Although it does not improve the expressive power of the approach, it does facilitate the implementation of some Execution Functions by allowing Execution Functions to rely, in a concurrency-aware manner, on other Execution Functions. Ultimately, this means that the code of the Execution Function implementations is more alike traditional programming which relies heavily on method calls to modularize and reuse code. This feature thus improves the language designer’s experience with the approach, by allowing them to rely on more traditional programming techniques in the metalanguage used for the Execution Function implementations. However, the main downside is that the control flow information must be specified twice: first in the implementation of ef_{caller}, and second in the MoCMapping. This means that any changes to either must also be ported to the other one.

3.8 Semantic Variation Points

In Chapter 2, we have defined and illustrated the notion of *Semantic Variation Points* (SVPs). They are language specification parts left intentionally under-specified to allow further language adaptation to specific uses. Using traditional language design techniques, their specification, implementation and managing is difficult. In this section, we present how the concurrency-aware approach facilitates the specification and implementation of SVPs for concurrency-aware xDSMLs. We draw the difference between a *Language* (the specification of a syntax and of a semantics that may contain SVPs) and its *Dialects* (which implement a language, making choices about some – possibly all – SVPs of the language). This work has been detailed and illustrated on Statecharts and its variants in [86].

3.8.1 Challenges

Tools commonly provide only one dialect, thus constraining the end-user to work with the selected specific implementation of SVPs, which may not be the best-suited for their needs. Besides, it also complicates the cooperation between tools, since they may implement SVPs differently, giving a different semantics to the same syntax. Two engineers with different backgrounds may also assume different meanings for the same model, which impairs communication. Finally, large projects may need to use several dialects cooperatively, which means that this issue cannot be simply reduced to the choice of a unique tool: one dialect with an associated tool may be the best fit for a particular aspect of a system, but other ones may be better-suited for other aspects of the system.

In the rest of this section, we will show how the modularity of the concurrency-aware approach towards the execution semantics of xDSMLs facilitates the specification and management of SVPs. We will also show how SVPs pertaining to the concurrency concerns can easily be implemented thanks to the separation of concerns of the approach. We will consider the SVPs of fUML [116]. In the fUML specification, the notions of time, communication and concurrency are delegated to the tool implementors. Tool vendors are thus responsible for specifying and documenting the implemented solution.

3.8.2 SVPs in Concurrency-aware xDSMLs

SVPs are usually specified informally, which makes their identification difficult. More often than not, the specification document describes all allowed possibilities, while a reference implementation defines the default implementation of SVPs. In the concurrency-aware approach, SVPs can manifest themselves in any part of the execution semantics. When

they are confined to only one of the concerns, the approach facilitates their specification and implementation since only one of the concerns is involved.

SVPs pertaining to the Semantic Rules

SVPs related to the runtime state or its evolution are contained in the Semantic Rules. Changing a Queue into a Stack, in order to implement a Last-In-First-Out policy instead of a First-In-First-Out, or incrementing a value twice instead of once to double a ressource consumption, are examples of such SVPs.

In the fUML specification [116], the guards of edges outgoing a DecisionNode may be evaluated in an arbitrary order, possibly in parallel. We could decide to use an arbitrary order, implemented in an Execution Function.

Implementations of such SVPs can be realized by overriding the corresponding Execution Data and Execution Functions. It is also often possible to prevent functions from being overriden (*e.g.*, in Java using the "final" keyword), allowing the language designer to ensure key parts of the semantics cannot be modified.

SVPs pertaining to the Communication Protocol

The fUML example mentioned above can also be realized by implementing several arbitrary orders in different Execution Functions, and then defining in the Mapping Protocol which one to use. More generally, variations of the Communication Protocol can be used to create dialects based on the same MoCMapping and Semantic Rules. In particular, the Feedback Protocol (as presented in Section 3.6) can also change to modify the semantics.

SVPs pertaining to the MoCMapping

The most interesting aspect of the approach for SVPs however, lies in the MoCApplication, and by extension, in the MoCMapping. Since it captures the concurrency concerns based on a symbolic partial ordering, it specifies explicitly all allowed control flow possibilities. Each dialect can remove the execution paths that do not correspond to its intended semantics by further restraining the symbolic partial ordering (*i.e.*, if expressed using constraints, by specifying additional constraints in the MoCMapping). In fact, nondeterminisms in the concurrency model (resulting in conflicts in the Event Structure) can all be seen as potential SVPs.

However, it is possible that some nondeterminisms are part of the execution semantics of the language (the language is thus indeterministic by intention), or that they should instead be solved by the platform on which the language is deployed (*i.e.*, the runtime

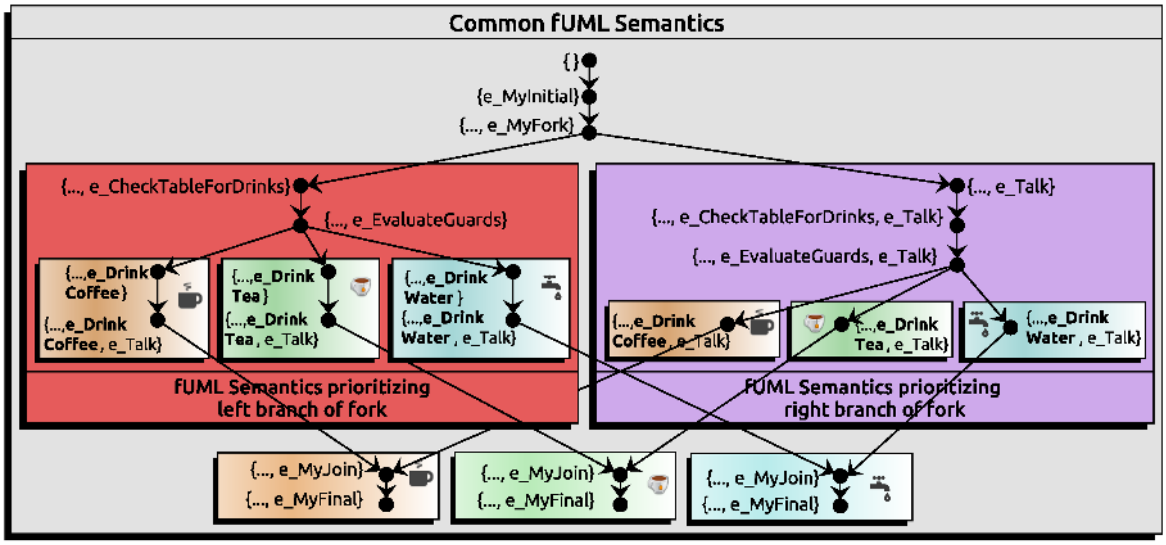

Figure 3.35: Simplified Event Structure illustrating a Semantic Variation Point of fUML.

of the concurrency model is specialized for a specific execution platform). Therefore, the MoCMapping metalanguage must provide the means to hinder some parts of its symbolic partial ordering from being specialized further.

Figure 3.35 illustrates a possible Semantic Variation Point of fUML. The branches of a ForkNode can be executed in any order. In particular, one can choose to execute the branches from left to right, or from right to left. In this figure, there is a "common semantics" captured by the initial MoCMapping, representing the execution semantics as given in the fUML specification [116]. Two different implementations are illustrated: one where we first execute the left branch of the ForkNode (the drinking part of the activity of Figure 3.2), and another one where we first execute the right branch of the ForkNode (the talking part of the activity). They can be implemented simply by extending the original MoCMapping and specifying additional constraints that result in the event structures we have shown.

The difference between specializing a language for a specific environment and implementing a Semantic Variation Point is blurry. SVPs sometimes represent adaptation points for a specific platform (distributed, highly parallel, etc.). Both are implemented by specializing the MoCMapping used to define the concurrency concerns of the xDSML. To better manage these SVPs, they can be implemented in a modular way so that dialects are then conceived by merging specific SVP implementations; similar to creating a new class in Aspect-Oriented Programming by extending an existing class and weaving existing aspects onto it.

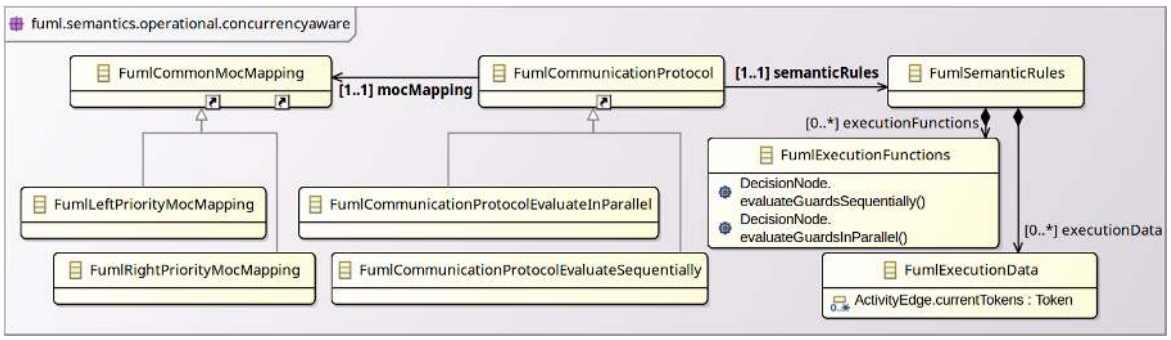

Figure 3.36: Metamodel representing the structure of the concurrency-aware operational semantics of fUML with its different variations.

For instance, based on the two fUML SVPs we have used as examples (pertaining to the order of evaluation of the guards of a DecisionNode and to the order of execution of the branches of a ForkNode), we can create as many dialects as the product of the number of implementations for the first SVP and of the number of implementations for the second SVP. The MoCMapping metalanguage and the code organization of the SVP implementations should enable the creation of dialects thanks to a cherry-pick of SVP implementations.

Finally, Figure 3.36 shows, as a metamodel, the semantics of fUML with its different variations. In the Semantic Rules, different Execution Functions are implemented for the different strategies of evaluation of the guards of a DecisionNode. Which one is used depends on the implementation of the Communication Protocol of fUML used. For the concurrency concerns, there is a common MoCMapping, which can be used as such (*i.e.*, the heuristic of the runtime will be in charge of determining how concurrent branches are executed), or extended with additional constraints to implement a particular strategy.

3.8.3 Feature Summary

This feature is a direct benefit of the separation of concerns induced by the concurrency-aware xDSML approach. It provides a sound and practical manner to specify and implement SVPs, particularly when they are related to the concurrency concerns of an xDSML. Managing semantic variants of xDSMLs is often difficult because a change in the execution semantics spreads through a lot of resources (*i.e.*, specifications as models, code, functions, etc.). Thanks to the separation of concerns, many SVPs can be confined to only one of these aspects. Then, depending on the extension mechanisms provided by each aspect's metalanguage, SVPs can be implemented more naturally. In the case of concurrency-related

SVPs, partial orders in Event Structures provide a very practical means for such extensions (*i.e.*, by refining the partial order through the definition of additional constraints), ultimately facilitating the management of the different dialects of an xDSML.

3.9 Concurrency-aware xDSMLs for Reactive Systems

Modern highly-concurrent systems are often reactive, in the sense that they must be able to react to the occurrence of some form of external event. This is most commonly the case for autonomous systems, whose purpose is precisely to be able to function without human intervention. In that context, determining the natures of the possible inputs and ensuring the correct behavior of the system for all possible inputs is one of the main aspects of reactive systems design. We propose to extend the concurrency-aware approach in order to enable the specification of xDSMLs aimed at specifying reactive systems.

3.9.1 Purpose

In the approach we have described so far, the concurrency model specifies all possible execution scenarios, leaving little room for the representation of incoming events. Representing all possible inputs in the concurrency model makes it complex, and usually involves representing parts of the data in the concurrency model, which defeats the initial objective of the concurrency-aware approach.

A possible workaround consists in regularly calling an Execution Function whose role is to check for some external input (via arbitrary code in the Execution Function implementation). This mechanism remains opaque, relying on implicit connections made in the Execution Function implementation, and on side-effects on the runtime state of the model.

Moreover, implementing data flows between Execution Functions is difficult because the MoCMapping is data-independent: as such, it cannot take into account the possible parameters of the Execution Functions. This was one of the motivations for the definition of Composite Execution Functions, presented in Section 3.7, which come with problems of their own. A workaround for this issue consists in storing the data that need to flow into the fields of a model element accessible by both ends of the flow, but this, too, relies on side-effects and implicit design rules.

We propose to augment the concurrency-aware approach with the means to specify reactive systems. This feature has two main aspects. First, we must be able to take into account external parameters during the execution of a model, using an explicit and dedicated mechanism (*i.e.*, unlike the workaround mentioned above). We propose to do this

by defining Mappings with parameters, similar to how in programming, functions have parameters. Then, one of the main use of these parameters is to be able to exploit them in the Execution Functions. Therefore we propose to provide the means to specify Execution Functions with parameters. We study the specification and runtime aspects of both issues.

3.9.2 Challenges

The first challenge is the specification of the parameters for Mappings and Execution Functions. In particular, we will focus on the compatibility between the type systems used in the respective metalanguages for the Communication Protocol and the Semantic Rules.

Then, we must consider the changes to the runtime. We must determine how arguments are provided to the MappingApplications, respectively to the Execution Function calls. For the latter, we will describe how the Composite Execution Function feature presented in Section 3.7 is managed.

3.9.3 Illustrative Example

The fUML example Activity is initially not reactive. We propose to modify the `Action` nodes such that their execution requires a String parameter. This way, the "CheckTableForDrinks" node returns, through its OutputPin, the given String, corresponding to the drink found on the table, instead of choosing randomly between "Coffee", "Tea" and "Water".

3.9.4 Defining Parameters for Execution Functions

Specification

The definition of Execution Functions with parameters should be done in the usual way of the metalanguage for the Semantic Rules. This is typically done in the type signature of the function, by specifying the name and type of parameters. Listing 3.18 shows an example Execution Function definition with a String parameter in Java.

Listing 3.18: Defining the Execution Function for the fUML `Action` node, in Java, with a String parameter.

```
class Action extends ExecutableNode{
  // ...
  void execute(String inputString){
   // ...
  }
}
```

Runtime

At runtime, the arguments of an Execution Function call are provided by the Executor. Since the Executor is controlled by the Execution Engine, it is ultimately responsible for providing the arguments to the Executor. In the rest of this section, we will show how the engine originally obtains the arguments that are passed to the Execution Function calls.

3.9.5 Introducing Parameters in Mappings

Specification

Parameters of Mappings are specified thanks to a type and a name, just like for Execution Functions.

Let us suppose in the case of the example fUML Activity, we want the argument of the node "CheckTableForDrinks" to be provided at runtime. In that case, the corresponding Mapping must be defined with a parameter. Listing 3.19 shows the pseudo-code specification of that mapping.

Listing 3.19: A Mapping with a parameter, corresponding to the execution of an fUML `Action` node, specified using pseudo-code.

```
Mapping ExecuteActionNode(String inputString):
  upon et_executeActionNode
  triggers Action.execute(inputString)
```

In this listing, the Mapping "ExecuteActionNode" has a String parameter. Parameters of Mappings can be used for several purposes. In our case, we use it as the argument of the Execution Function called, `execute(String)`. If the Mapping has an associated

Feedback Policy (as defined in Section 3.6), then the parameters can also be used in it. For instance, we may want to compare the String argument of the Mapping with the String value returned by an Execution Function. Listing 3.20 shows such a Feedback Policy.

Listing 3.20: Using the parameter of a Mapping in its Feedback Policy, in pseudo-code.

```
1 Mapping MyMapping(String someString):
2   upon someMoCTrigger
3   triggers MyClass.myExecutionFunction() returning resultString
4   feedback:
5     resultString == someString => allow MyClass.someOtherMoCTrigger
6     resultString != someString => allow MyClass.anotherMoCTrigger
```

In that example, the impact on the control flow of the value returned at runtime by the Execution Function `myExecutionFunction` will depend on its comparison to the String passed as argument to the corresponding Mapping.

Similar to what was discussed in the section about the specification of the Feedback Policy (Section 3.6), the type systems used must be compatible. This means that the metalanguages for the Semantic Rules, Mapping Protocol and Feedback Protocol must have a common type system, or at least a way to communicate type informations. This means they must all originally integrate the metalanguage for the abstract syntax of the xDSML, since it is likely that it provides a decent basis.

Figure 3.37 shows an excerpt from the metamodel of the Communication Protocol extended with parameters for the Execution Functions and the Mappings.

Runtime

Part of the runtime is straightforward. If the arguments of the Mappings are to be used by the Execution Function, then they are passed to it during the launching of the Execution Function. If they are to be used in the Feedback Policy then they are used during the interpretation of the Feedback Policy once the associated Query has returned a value (*cf.* Section 3.6).

The complex part of the runtime is how arguments are provided to the Mappings in the first place. When handling an execution step, the runtime first retrieves the set of possible Scheduling Solutions from the Solver, and an heuristic chooses one of them. It is then matched against the Communication Protocol Application to deduce which Execution Function calls must be executed. The arguments of a MappingApplication must

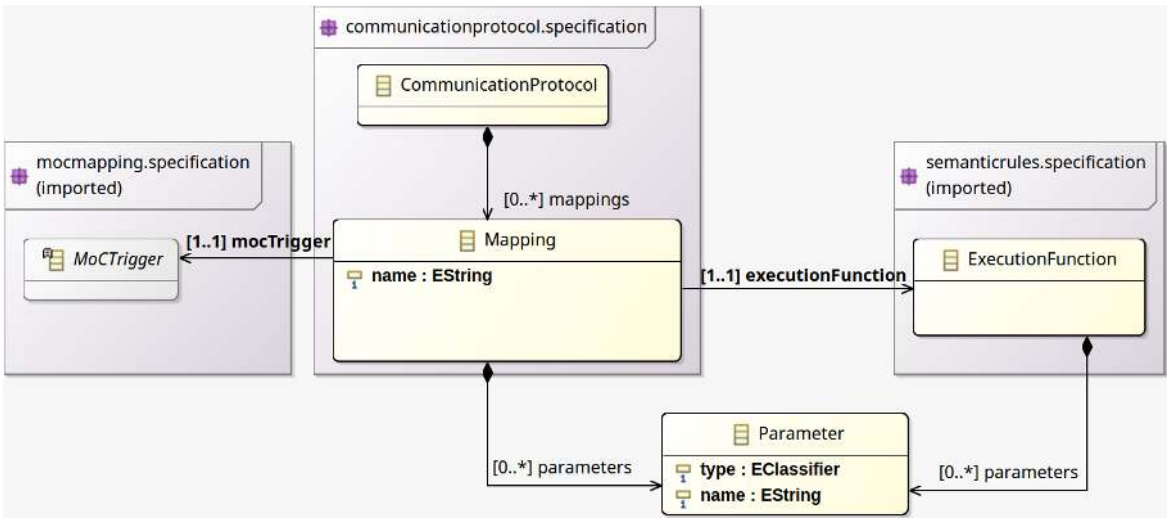

Figure 3.37: Excerpt from the metamodel of the Communication Protocol showing its extension with parameters for Mappings and Execution Functions.

thus be provided when selecting the Scheduling Solution leading to it. This means that the heuristic is responsible for providing the arguments of the Mappings.

This makes the implementation of the heuristic complex. Indeed, the heuristic is generic and not tied to a particular domain. However, parameters are typically domain-specific types and values. This means that the language designer must provide parts of the heuristic implementation used, in order to enable the end-user to enter valid arguments for the MappingApplications of a system.

This situation is a bit similar to designing the concrete syntax of a language, as some constraints are difficult to capture in the abstract syntax of the language. In Xtext [7], this is concretized as the *scoping mechanism*[7]. It allows the definition of a scope for each part of the model creation. In other words, it defines rules guiding the creation of a valid program (instance of the abstract syntax). Such a mechanism could be used in order to guide the end-user in selecting valid arguments, for instance via a Graphical User Interface or a tool-supported Command-Line Interface (*e.g.*, with auto-completion features, etc.).

Unfortunately, this sort of mechanism works well only when the possible values are already known by the model at runtime, *e.g.*, if they are parts of the model or part of an enumeration. Creating objects for the purpose of using them as the arguments of MappingApplications would require an event more complex definition of the heuristic, or the use of an external program through an API to create the expected complex object. IDE features, like those generated by Xtext for the Eclipse platform, implement this through

[7]https://eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping

the notion of *template proposals*. It is essentially a syntactic help dedicated to the creation of new constructs (*i.e.*, objects, methods, etc.). Figure 3.38 shows the auto-completion (fed by the scoping mechanism we have described above) and the template proposals features in the Eclipse IDE for Java (Java Development Tools – JDT).

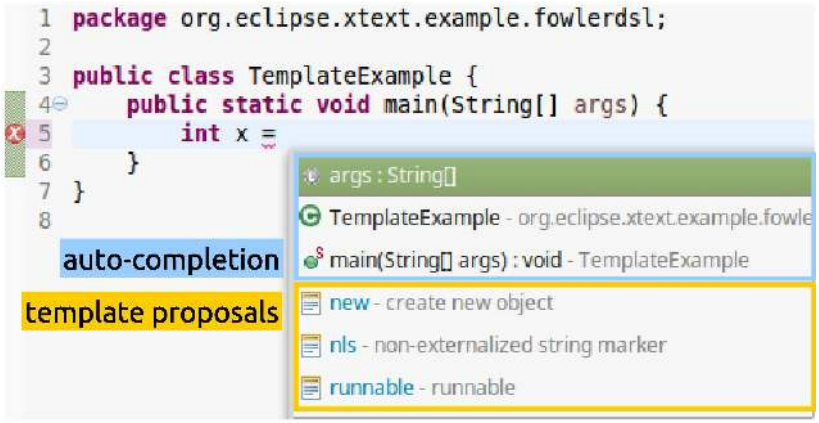

Figure 3.38: Auto-completion and template proposals feature in the Eclipse IDE for Java.

3.9.6 Feature Summary

This feature extends the concurrency-aware xDSML approach with the means to support xDSMLs with parameterized language constructs. This could previously be done implicitly by relying on side effects inside the model; this feature makes this explicit in the execution semantics model. The behaviors (*i.e.*, the Mappings) can be parameterized during both directions of the communication: either by parameterizing the Execution Functions, or the Feedback Policies. Such parameterized behaviors are required when the environment of the xDSML is not captured as part of the xDSML (*i.e.*, a user input, some external data, etc.).

3.10 Behavioral Interface of Concurrency-aware xDSMLs

Previous features focused on facilitating or enabling the specification of the execution semantics of concurrency-aware xDSMLs. In this section, we take a step back to consider the use that can be made of concurrency-aware xDSMLs. We argue that the Mappings of an xDSML can be considered as the *behavioral interface* of the language, *i.e.*, that it represents the behavior of the language and can be exploited by other programs or languages.

3.10.1 Purpose

By expressing the communication between the Execution Functions and the MoCTriggers, the Mappings of an xDSML give a mapping of abstract actions, scheduled by a partial ordering, with concrete actions. As such, Mappings represent the interface of the language constructs behaviors, *i.e.*, they represent a high-level view of what happens in the model (*i.e.*, the actual Execution Function implementation). They can also be the subject of external components, within the limits of the partial ordering defined by the MoCMapping, *i.e.*, an external program may be used as the heuristic of the runtime and provide decisions based on which Mappings are occurring in the possible solutions. In that sense, Mappings represent an interface which can be both listened to (*i.e.*, which Mappings are being executed) and spoken to (*i.e.*, by making arbitrary decisions between the possible Scheduling Solutions, so indirectly between sets of Mapping occurrences).

External components may want to interact with a concurrency-aware xDSML (or, punctually, with a concurrency-aware executable model) for two purposes. The first one is to control which Mappings are occurring, by implementing the heuristic of the execution engine. In that case, it is also in charge of providing the arguments to the Mappings, if there are some, *cf.* Section 3.9. The second one consists in observing which Mappings are executed. This allows for a fine-grained observation of the behavior of the language, which can be exploited for instance to represent the execution (as a trace of Mappings), to coordinate the execution of other executable models, etc. In the context of GEMOC project, this interface is used for the coordination of several concurrency-aware xDSMLs [153].

In order to cater to these two purposes, we propose to augment the specification of the Mappings with two features. The first one is the possibility to define the *visibility* of Mappings, enabling the language designer to explicitly separate the Mappings intended to be observed (*i.e.*, because they are the most relevant, or more practically, if there are many of them, the interface becomes cluttered) from the other Mappings (possibly required for technical reasons, or representing low-level details of the execution). For instance in fUML, the most relevant behavior, for an external observer, is usually the execution of a node. Whereas the execution of the guards outside a `DecisionNode` are considered as internal details of the model. The second one is the possibility to define patterns of Mappings, effectively defining what we call *Composite Mappings*, which provide a more abstract, higher-level view of the behavior of a language or model. Abstracting away unnecessary details of the language's execution facilitates its use by providing a more adequate conceptual (and programmatic) representation to the user, or to other programs.

3.10.2 Challenges

We must first identify how many different visibilities there can be, and what they correspond to, with respect to both interface roles played by the Mappings. Their specification should be added in the metalanguage of the Communication Protocol.

For the Composite Mappings, there is first the question of understanding what they represent. We will then present how their specification and runtime can be done, although these challenges are very implementation-dependent. We will also give some examples based on our implementation in order to illustrate their purpose.

3.10.3 Mapping Visibility

Specification

We have identified the need for only two types of visibility, which we call `public` and `private`.

Mappings are public by default. This means that when they occur during the execution of a model, they are published as occurring, thus external components observing the execution of the model know about it. By contrast, private Mappings are not published as occurring during the execution, as they are not meant to be shown to external elements.

As an example, we may consider that during the execution of fUML models, the only relevant event is the execution of the ActivityNodes. Therefore, the internal mechanic of evaluating the guards after a DecisionNode, etc. should not be published. Listing 3.21 shows an excerpt from the pseudo-code specification of the Communication Protocol for fUML where the Mapping corresponding to the execution of a node is public, but the one corresponding to the evaluation of a guard is private.

Listing 3.21: Excerpt from the Communication Protocol of fUML, specified using pseudo-code, with visibility added to the Mappings definition.

```
1  public Mapping ExecuteActivityNode:
2    upon et_executeNode
3    triggers ActivityNode.execute()
4
5  private Mapping EvaluateGuard:
6    upon et_evaluateGuard
7    triggers ActivityEdge.evaluateGuard() returning result
8    feedback:
9      result = true => allow ActivityEdge.et_mayExecuteTarget
10     result = false => allow ActivityEdge.et_mayNotExecuteTarget
```

The visibility of a Mapping does not influence its interaction with the heuristic of the runtime. Otherwise, it would defeat the purpose of the heuristic.

Figure 3.39 shows an excerpt from the metamodel of the Communication Protocol extended with the feature we just described.

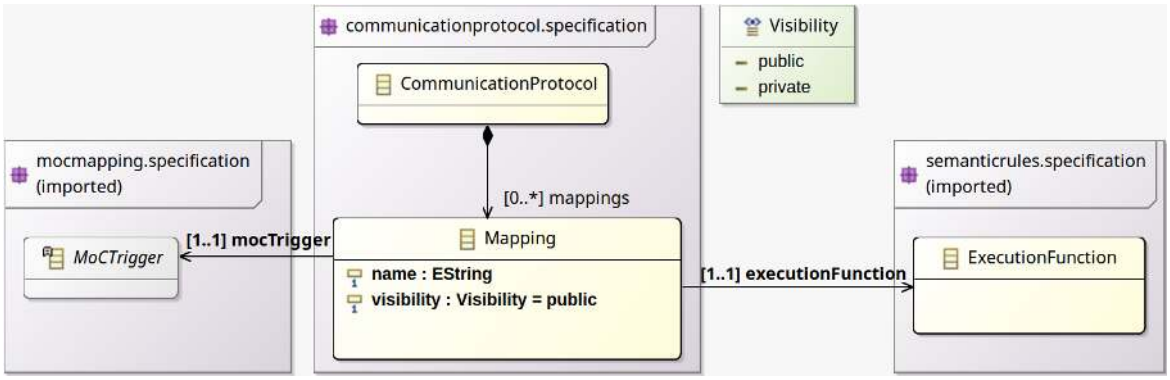

Figure 3.39: Excerpt from the metamodel of the Communication Protocol with the notion of Visibility for the Mappings.

Runtime

For every execution step, the Execution Engine publishes the collection of Mappings that are executed during this step (and with which arguments). To implement the visibility feature, it should not add the private Mappings to this collection.

3.10.4 Composite Mappings

Specification

Composite Mappings are specified alongside the other Mappings, thanks to some form of "pattern" over other Mappings. For instance, the execution of the example fUML Activity can be considered as completed if we first observe that its `InitialNode` has been executed, and then that its `FinalNode` has been executed. The pattern of first executing the InitialNode and then the FinalNode delimits the execution of this Activity. In terms of patterns of Mappings, this means that we want to first observe an occurrence of the MappingApplication "ExecuteNode_MyInitial" and then observe an occurrence of the MappingApplication "ExecuteNode_MyFinal". Another example is if, in an Activity, part of it can be executed several times. Then we may want to be able to observe when that part has been executed a certain number of times, "n". Such a Composite Mapping can be

defined as “n” consecutive occurrences of the same Mapping. In particular, we may want to observe, say, 5 times the execution of the example Activity, in which case the Composite Mapping is defined through a pattern of another Composite Mapping.

We propose to reify such patterns as the *Composite Mappings*: Mappings which are defined thanks to a pattern over previously-defined Mappings. These Mappings *do not* map a MoCTrigger to an Execution Function ; they merely correspond to a behavioral pattern of the language.

These Mappings may be specified as ’private’, for instance if they are only used for the definition of other composite Mappings. They may also have parameters, for instance if we want to define a pattern that is valid only if the occurrences occur with certain arguments (*cf.* Section 3.9).

The patterns that can be defined depend entirely on the expressive power provided by the metalanguage for the Communication Protocol. In particular, it can provide the means to define *libraries* of patterns for the definition of Composite Mappings. Later, we illustrate some core patterns we have identified for the implementation of this feature.

Runtime

To implement this feature, the runtime for the Communication Protocol, the Matcher, must be updated adequately.

It must first match the selected Scheduling Solution against the specification of the non-Composite Mappings, in order to deduce which Mappings are occurring. Based on this, it must then match the Composite Mappings which are occurring in an incremental manner, so that composites which are defined thanks to other composites can be matched correctly. As we will detail later, this step may require the use of the previous Execution Steps, as composites may span over several steps.

Occurrences of Composite Mappings do not have an impact on the state of the model, unlike regular Mappings which trigger an Execution Function. Composites merely serve to provide a higher-level, abstract view of the behavior of the xDSML. Their occurrences do not trigger any change in the model. Additionally, patterns may only be fully identified once the last element of the pattern has occurred ; otherwise it would mean speculating about the future of the execution. Therefore, Composite Mappings always occur in coincidence with the Mapping occurrence which completed the pattern.

There are two possible variations in the implementation of the runtime. To illustrate these two versions, we will consider two patterns based on the example fUML Activity. The first one corresponds to an occurrence of “ExecuteNode_MyFork” followed by an oc-

currence of "ExecuteNode_CheckTableForDrinks". The second one corresponds to an occurrence of "ExecuteNode_MyFork" followed by an occurrence of "ExecuteNode_Talk".

In the first variation, Mapping occurrences are consumed upon occurrence of a composite whose pattern relies on them. In our example, this means that since the ForkNode "MyFork" will only ever be executed once, both composites are intrinsically exclusive. In the second variation, occurrences are not consumed, allowing for both composites to occur. However this raises an additional challenge when multiple occurrences of the same Mapping may occur. In that case, suppose "ExecuteNode_MyFork" occurs twice. Should the composites defined above use these two occurrences, or only the latest?

We leave these issues open for implementors, but they should be documented profusely since they fundamentally affect the semantics of Composite Mappings. They may also be reified in the Communication Protocol metalanguage, but this complexity may render the definition of Composite Mappings even more difficult for the language designer.

Examples

Let us consider a few examples of patterns. Listing 3.22 shows a pseudo-code specification of two Mappings, `MappingA` and `MappingB`. The simplest patterns are called *instantaneous*, *i.e.*, they occur over a single step of execution. Following are three such patterns:

- *Coincidence of A and B*: when both A and B occur.
- *Disjunction of A and B*: when A occurs or B occurs.
- *Exclusive Disjunction of A and B*: when A or B, but not both at the same time, occur.

Listing 3.23 shows an example specification using these patterns, in pseudo-code.

Listing 3.22: Example Communication Protocol specification, in pseudo-code.

```
1 Mapping MappingA:
2   upon mocTriggerA
3   triggers MyClass.executionFunctionA
4
5 Mapping MappingB:
6   upon mocTriggerB
7   triggers MyClass.executionFunctionB
```

Listing 3.23: Example specification of Composite Mappings with instantaneous patterns, in pseudo-code.

```
1 Composite Mapping CompositeAandB:
2   MappingA and MappingB
3
4 Composite Mapping CompositeAorB:
5   MappingA or MappingB
6
7 Composite Mapping CompositeAxorB:
8   MappingA xor MappingB
```

The following description of the runtime is at the language-level (for readability purposes) but in reality it applies at the model-level.

1. mocTriggerA occurs:
    - MappingA occurs.
    - CompositeAorB and CompositeAxorB occurs.
2. mocTriggerB occurs:
    - MappingB occurs.
    - CompositeAorB and CompositeAxorB occurs.
3. mocTriggerA and mocTriggerB occurs:
    - MappingA and MappingB occur.
    - CompositeAorB and CompositeAandB occur.

These basic patterns are similar to well-known logical operations, as within an execution step, either an event occurs or it does not occur. More complex instantaneous patterns may be devised and proposed by the Communication Protocol metalanguage, based on the experience we have of logical operations.

More interesting is the possibility to define patterns over several execution steps. These patterns are called *non-instantaneous* patterns. They cannot be compared with logical operations, as reasoning on the *absence* of an event in a context of partial ordering does not make sense. Therefore, these patterns may span over a lot of execution steps. For instance, the pattern we have mentioned earlier, corresponding to the execution of the InitialNode

and then the execution of the FinalNode of the example fUML Activity, spans over the whole execution of the model (10 execution steps in our implementation, *cf.* Appendix C).

We will consider the two following patterns as examples of non-instantaneous patterns:

- *Sequence of A and B*: when A has occurred, and B occurs in the current step. There may be an indefinite number of other steps between the one containing the occurrence of A and the one containing the occurrence of B.
- *n-Iteration of A*, with $n \in \mathbb{N}^+$: when A has occurred $n-1$ times and A occurs in the current step. This may span over an indefinite number of steps as well.

Listing 3.24 shows an example specification using these patterns, in pseudo-code.

Listing 3.24: Example specification of Composite Mappings with instantaneous patterns, in pseudo-code.

```
1 Composite Mapping CompositeAthenB:
2   MappingA -> MappingB
3
4 Composite Mapping CompositeThreeAs:
5   MappingA[3]
```

With these mappings, the implementation choices mentioned in the description of the runtime are key elements of the semantics given to these patterns. For instance, if we consider the three following execution steps:

Step 1 : occurrence of MappingA

Step 2 : occurrence of MappingA

Step 3 : occurrence of MappingA and MappingB (Current Step)

There are 4 possible outcomes, depending on the implementation choices realized:

1. If patterns can only refer to the latest occurrences of mappings, and used mappings are consumed upon match:
   Occurrence of *CompositeThreeAs* (with MappingA occurrences from Steps 1, 2 and 3)
   OR
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 2, Mapping B occurrence from Step3).

2. If patterns can only refer to the latest occurrences of mappings, and used mappings are *not* consumed upon match:
   Occurrence of *CompositeThreeAs* (with MappingA occurrences from Steps 1, 2 and 3)
   AND
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 2 and MappingB occurrence from Step 3).

3. If patterns can use all occurrences of mappings, and used mappings are consumed upon match:
   Occurrence of *CompositeThreeAs* (with MappingA occurrences from Steps 1, 2 and 3)
   OR
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 1 and MappingB occurrence from Step 3)
   OR
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 2 and MappingB occurrence from Step 3).

4. If patterns can use all occurrences of mappings, and used mappings are *not* consumed upon match:
   Occurrence of *CompositeThreeAs* (with MappingA occurrences from Steps 1, 2 and 3)
   AND
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 1 and MappingB occurrence from Step 3)
   AND
   Occurrence of *CompositeAthenB* (with MappingA occurrence from Step 2 and MappingB occurrence from Step 3).

These examples show the impact the choices made in the implementation of the metalanguage may have. Depending on the Communication Protocol metalanguage, more complex non-instantaneous patterns may be devised and proposed, enabling the definition of complex Composite Mappings.

Customizing the Model-level Generation of Composite Mappings

We have described earlier how the model-level specifications (MoCApplication, Communication Protocol Application, Semantic Rules Calls) are obtained based on the language-level specifications (MoCMapping, Communication Protocol, Semantic Rules). This step is also called the "unfolding" since it mainly consists in considering a concept and generating each concern's equivalent specification for each instance of that concept.

So far, the unfolding of our example Composite Mappings has been straightforward as, for the sake of example, we only considered one concept and one model element (instance of that concept). For instance, consider the Composite Mapping defined as per Listing 3.25.

Listing 3.25: Example specification of the Composite Mapping `CompositeAandB`, in pseudo-code.

```
1 Composite Mapping CompositeAthenB:
2   MappingA -> MappingB
```

MappingA and MappingB being defined in the same context (MyClass), their unfolding results in as many Composite MappingApplications as there are instances of MyClass in the model. For example, if the model has two instances of MyClass, Object1 and Object2, the resulting Communication Protocol Application is as shown on Listing 3.26.

Listing 3.26: Example specification, in pseudo-code, of the Composite Mapping `CompositeAandB`.

```
1  MappingApplication MappingA_Object1:
2    upon mocTriggerA_Object1
3    triggers Object1.executionFunctionA
4
5  MappingApplication MappingB_Object1:
6    upon mocTriggerB_Object1
7    triggers Object1.executionFunctionB
8
9  MappingApplication MappingA_Object2:
10   upon mocTriggerA_Object2
11   triggers Object2.executionFunctionA
12
13 MappingApplication MappingB_Object2:
14   upon mocTriggerB_Object2
15   triggers Object2.executionFunctionB
16
17
18 Composite MappingApplication CompositeAthenB_Object1:
19   MappingA_Object1 -> MappingB_Object1
20
21 Composite MappingApplication CompositeAthenB_Object2:
22   MappingA_Object2 -> MappingB_Object2
```

In order to be able to define more complex Composite Mappings, we propose to add a specification alongside the Composite Mapping definition called the *Unfolding Strategy*. It defines the strategy that the translator must use when unfolding the Composite Mapping down to the model-level. Listing 3.27 shows the specification, in pseudo-code of an example unfolding strategy. "<>" is the "not equal" operator in this pseudo-code.

Listing 3.27: Example unfolding strategy for the Composite Mapping `CompositeAandB`, in pseudo-code.

```
1 Composite Mapping CompositeAthenB:
2   with {
3     o1 : MyClass,
4     o2 : MyClass
5   } where {
6     o1 <> o2
7   }
8   MappingA(o1) -> MappingB(o2)
```

With this strategy, the composite MappingApplications resulting of CompositeAthenB will be very different from the default result that would have been obtained, because the MappingApplications used in the pattern in place of MappingA and MappingB will never be in the context of the same model element.

Listing 3.28 shows the resulting Communication Protocol Application.

This mechanism allows the definition of complex Composite Mappings where relations between the contexts of the Mappings used in the pattern are guaranteed.

For instance, our initial example of Composite Mapping was to represent the full execution of an fUML Activity. Such a mapping can be specified as shown on Listing 3.29. Listing 3.30 shows another way to specify the same behavior.

Listing 3.28: Composite Mapping Applications resulting from the unfolding strategy specified on Listing 3.27, in pseudo-code.

```
1 Composite MappingApplication CompositeAthenB_Object1_Object2:
2   MappingA_Object1 -> MappingB_Object1
3
4 Composite MappingApplication CompositeAthenB_Object2_Object1:
5   MappingA_Object2 -> MappingB_Object2
```

Listing 3.29: Example of Composite Mapping capturing the full execution of an fUML Activity.

```
1 Composite Mapping FullActivityExecution:
2   with {
3     initialNode : InitialNode,
4     finalNode : FinalNode
5   } where {
6     true
7   }
8   ExecuteActivityNode(initialNode)
9     -> ExecuteActivityNode(finalNode)
```

Listing 3.30: Alternative manner of specifying the example Composite Mapping capturing the full execution of an fUML Activity.

```
1 Composite Mapping FullActivityExecution:
2   with {
3     initialNode : ActivityNode,
4     finalNode : ActivityNode
5   } where {
6     initialNode kindof InitialNode,
7     finalNode kindof FinalNode
8   }
9   ExecuteActivityNode(initialNode)
10    -> ExecuteActivityNode(finalNode)
```

The former example listing relies on polymorphism when specifying the pattern of Mappings (*i.e.*, the Mapping `ExecuteActivityNode` is defined for ActivityNode so it is applicable for its subtypes), whereas the latter example listing relies on the metalanguage providing the `kindof` operator.

Composite Mappings with Parameters

Finally, we want to illustrate the definition of Composite Mappings with parameters. Since Composite Mappings are just patterns based on previously-defined Mappings, their occurrences do not require the insertion of argument values by the heuristic of the runtime, as we have described in Section 3.7. Instead, it is at the matching stage of the composites that we are looking for particular argument values in the Mappings that have occurred.

Listing 3.31 shows the definition of a Composite Mapping with parameters.

Listing 3.31: Example specification of a Composite Mapping with parameters, in pseudo-code.

```
1  Mapping MappingA(Type1 x):
2    upon mocTriggerA
3    triggers MyClass.executionFunctionA(x)
4
5  Mapping MappingB(Type2 y):
6    upon mocTriggerB
7    triggers MyClass.executionFunctionB(y)
8
9  Composite Mapping CompositeAthenB(Type1 x, Type2 y):
10   MappingA(x) -> MappingB(y)
```

This Composite Mapping occurs with the arguments "x" and "y" when:

- MappingA occurred, and in the current step, MappingB occurs.
- MappingA occurred with "x" as an argument.
- MappingB occurs with "y" as an argument.

Patterns may also be defined based on specific expected argument values. For instance, consider the Composite Mapping specified using pseudo-code on Listing 3.32.

Listing 3.32: Example specification of a Composite Mapping with expected argument values, in pseudo-code.

```
1  Mapping MappingPrint(String s):
2    upon someMocTrigger
3    triggers MyClass.executionFunctionPrint(s)
4
5  Composite Mapping CompositePrintHelloThenPrintWorld:
6    MappingPrint("Hello") -> MappingPrint("World")
```

This mapping occurs when the Mapping "MappingPrint" first occurs with the argument "Hello" and then with the argument "World".

Related Work

The feature we just described is similar to what is known as *Complex Event Processing* (CEP) [91]. There exists many technologies for CEP, depending on the technical ecosystem considered, among which Esper [40] (Java, .NET); Microsoft's StreamInsight [99] (.NET); Oracle Complex Event Processing [119] (Java); WSO2's Complex Event Processor [161] (Java); JBoss's Drools Complex Event Processing [70] (Java); Apache's Storm [3] and Flink [2] (Java). Unfortunately, no formal standard exists [122]. In some situations, the language used for specifying the CEP has an SQL-based syntax.

The context of these technologies is very different from the context of our work. They often focus on the execution performance of their runtime in order to achieve near real-time recognition of patterns of events. In our case, the most important feature is the expressive power of the metalanguage for the definition of patterns of events. In "Processing Flows of Information: From Data Stream to Complex Event Processing" [23, Section 3.8.2], the authors provide a complete list of operators found during the analysis of Information Flow Processing Systems, including approaches related to Complex Event Processing. They can be used as a basis for the implementation of this feature.

3.10.5 Feature Summary

This feature focuses on the nature of the Mappings of the Communication Protocol which represent the behavioral interface of the language. It does not affect directly the execution semantics of an xDSML, instead it merely changes how the xDSML's semantics is represented from an external point of view (*e.g.*, the user or another program). It is motivated by further uses of concurrency-aware xDSMLs in the GEMOC Project, where several xDSMLs are coordinated through operators exploiting their behavioral interfaces. In short, this feature participates in making possible the specification of a higher-level behavioral interface for concurrency-aware xDSMLs, in order to present a particular interface to other programs or languages. For example, it can be used to de-clutter the interface from technical details of the implementation, or from parts of the behavior that should not be visible (or are of no interest) for any potential external program.

3.11 Implementation

We describe the implementation of the concurrency-aware approach in the Eclipse-based GEMOC language workbench. It includes the description of the metalanguages provided

by the language workbench for the definition of the different concerns. The full source code of our implementation of fUML is provided in Appendix B.

3.11.1 Technical Space

The concurrency-aware approach has been implemented in an Eclipse-based application called the GEMOC Studio[8]. It is based on the Eclipse Modeling Framework (EMF) [36], the core component of the Eclipse Modeling Project[9]. These technologies have been presented in Chapter 2. EMF provides a large existing ecosystem of technologies and tools, including Java APIs, allowing the definition of metamodeling tooling using any JVM language. Moreover, the Eclipse Rich Client Platform (RCP) is a natural candidate for the development of a language workbench. Other platforms providing RCP abilities or metamodeling facilities exist, but the Eclipse platform is, so far, one of the strongest candidate when needing both at the same time. Its open-source nature and its licensing policy (Eclipse Public License[10]) also contribute to its adequacy.

The GEMOC Studio is an Eclipse application which embarks the metalanguages for the specification of concurrency-aware xDSMLs, as well as their runtimes. It also provides different facilities for the development, reuse and debugging of the different concerns composing an xDSML.

The studio is made up of two components:

The GEMOC Language Workbench, used to specify and edit concurrency-aware xDSMLs.

The GEMOC Modeling Workbench, used to create and execute models conforming to concurrency-aware xDSMLs.

Concurrency-aware xDSMLs defined thanks to the former can be automatically deployed in the latter, and benefit from generic execution and debugging facilities.

3.11.2 Metamodeling Facilities

EMF provides Ecore, an implementation of EMOF [112]. The Abstract Syntax of an xDSML can be specified as an Ecore metamodel. EMF provides several editors for EMF: tree-based, graphical and textual.

[8] http://gemoc.org/studio/
[9] http://www.eclipse.org/modeling/
[10] http://www.eclipse.org/legal/epl-v10.html

The associated static semantics can be expressed in terms of Object Constraint Language (OCL) invariants [113]. EMF provides its implementation of OCL as Eclipse OCL[11]. It includes editor and interactive consoles facilities. Both EMOF and OCL are standards from the Object Management Group (OMG)[12], as mentioned in Chapter 2.

Ecore provides facilities to automatically generate Java APIs for Ecore metamodels, enabling any JVM-based technology to exploit Ecore metamodels and models. We leverage this feature in the metalanguages implementations presented in the rest of this section.

3.11.3 Semantic Rules

To specify the Semantic Rules, the GEMOC Studio relies on the Kermeta 3 Action Language (K3AL) [32], which is built on top of Xtend [7] by INRIA (IRISA). K3AL allows the definition of *Aspects* for Ecore metaclasses, allowing us to define additional classes, attributes, references and operation implementations, specifying the Execution Data and Execution Functions. K3AL, just like Xtend, compiles into readable Java and provides an executor based on the Java Reflection API to dynamically execute the Execution Functions. Listing 3.33 shows the implementation of an Execution Function for ForkNodes using K3AL. The full Semantic Rules implementation for fUML is provided in Appendix B.

Listing 3.33: Excerpt from the Semantic Rules of fUML specified using Kermeta 3.

```
1  import org.gemoc.sample.fuml.ForkNode
2
3  @Aspect(className=ForkNode)
4  class ForkNodeAspect {
5    // Modifier
6    def public void execute() {
7      precondition(_self)
8      println("***␣ForkNode␣[" + _self.name + "]" + "***")
9      // Forks each incoming token and sends a version to each
       outgoing edge.
10     _self.outgoingEdges.forEach [ outgoingEdge |
11       outgoingEdge.currentTokens.clear()
12       _self.incomingEdges.forEach [ incomingEdge |
13         incomingEdge.currentTokens.forEach [ token |
14           if (token instanceof ObjectToken) {
15             val Object object = (token as ObjectToken).object
16             outgoingEdge.currentTokens.add(TokenHelper.
       createObjectToken(object))
```

[11] http://projects.eclipse.org/projects/modeling.mdt.ocl
[12] http://omg.org/

```
17            } else {
18              outgoingEdge.currentTokens.add(TokenHelper.
    createControlToken())
19            }
20          ]
21        ]
22      ]
23    }
24
25    // There should be at least one incoming edge with at least one
      token.
26    def private void precondition() {
27      val boolean atLeastOneIncomingEdgeHasAtLeastOneToken = !_self.
      incomingEdges.map [ incomingEdge |
28        incomingEdge.currentTokens
29      ].flatten.isEmpty
30        if (!atLeastOneIncomingEdgeHasAtLeastOneToken) {
31          throw new PreconditionException(_self)
32        }
33    }
34  }
```

In this example, we define and use a pre-condition for the Execution Function, as preconized in 3.3.4. This ensures that the Execution Function is called only in situations where it makes sense, facilitating the debug of the rest of the semantics during the development of the language.

More generally, the requirements for the Semantic Rules metalanguage are the following:

- Capacity to extend the Abstract Syntax with additional data (attributes, references, classes, etc.).
- Capacity to extend the Abstract Syntax with operation declarations and implementations.
- Runtime able to execute the operation implementations.

3.11.4 Model of Concurrency Mapping

In the GEMOC Studio, the MoCMapping is specified using the Event Constraint Language (ECL) [27], an extension of OCL enabling the definition of EventTypes (in the context of concepts from an Ecore metamodel) and of constraints between these EventTypes. ECL is

developed by INRIA (I3S). It provides a set of core constraints, facilitating the specification of complex symbolic partial orderings between the EventTypes. It follows the UML Profile for *Modeling and Analysis of Real-Time and Embedded systems* (MARTE) [110], standardized by the OMG[13]. Complex constraints can also be specified and capitalized into metamodel-agnostic libraries using MoCCML [29], developed by INRIA (I3S) and ENSTA Bretagne. Listing 3.34 shows an excerpt from the MoCMapping of fUML, specified using ECL. In this example, the EventType `moc_executeNode` is defined in the context of the concept `ActivityNode`. We then define a constraint to ensure that, for an edge, the source is generally executed before the target (except for `MergeNode` for which only one one of the incoming edges' source must have been executed).

Listing 3.34: Excerpt from the Model of Concurrency Mapping of fUML specified using the Event Constraint Language.

```
1  import 'platform:/resource/org.gemoc.sample.fuml.model/model/fuml.
       ecore'
2
3  package fuml
4   -- A node may be executed
5   context ActivityNode
6    def: moc_executeNode : Event = self
7
8   context ActivityEdge
9   -- In general, execute the source before the target.
10   inv executeSourceBeforeTarget:
11    ((self.guard = null) and
12     (not (self.targetNode.oclIsKindOf(MergeNode)))
13    ) implies(
14     Relation Precedes(self.sourceNode.moc_executeNode,
15                       self.targetNode.moc_executeNode)
16    )
17 endpackage
```

The full source code of the MoCMapping for our implementation of fUML is available in Appendix B.

More generally, the requirements for the Model of Concurrency Mapping metalanguage are the following:

- Capacity to specify the symbolic use of a Model of Concurrency (*i.e.*, so that the MoC is used for any model conforming to the xDSML).

[13]Object Management Group – http://www.omg.org/

- Generator to unfold the MoCMapping specification to any model conforming to the AS of the xDSML.

During the compilation phase, the MoCMapping defined in ECL is compiled into a *Clock Constraint Specification Language (CCSL)* [92] model. CCSL can be analyzed with a runtime called TimeSquare [28], which can generate execution traces. For the practical reasons mentioned in Subsection 3.6.4, during its execution, TimeSquare only provides the next set of possible configurations. TimeSquare provides Java APIs, allowing us to use it into our implementation.

Figure 3.40 shows an excerpt from the Value Change Dump (VCD) timing diagram of the execution of the example fUML Activity's MoCApplication. It represents the trace of the execution of an Event Structure by showing the occurrences of events as "ticks" of a "clock". In this figure are represented the events corresponding to the execution of the DecisionNode, respectively to the evaluation of the guards outgoing the DecisionNode. Figures 3.41, 3.42, and 3.43 show the VCD for the clocks corresponding to the result of the evaluation of the guard for each branch (respectively for coffee, tea and water), as explained in Section 3.6. The top line corresponds to the event "may..." while the bottom line corresponds to the event "mayNot...", for each branch. In this particular execution, coffee was found on the table so the branches for coffee and water are both allowed. Ultimately, only the branch corresponding to coffee will be executed (as per the fUML semantics [116]).

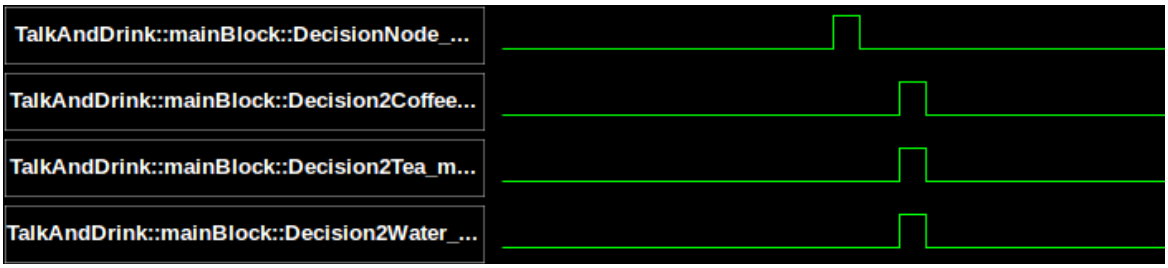

Figure 3.40: Excerpt from the trace of the execution of the MoCApplication of the example fUML Activity.

Figure 3.41: VCD for the events corresponding to allowing, respectively disallowing, the branch leading to drinking coffee.

Figure 3.42: VCD for the events corresponding to allowing, respectively disallowing, the branch leading to drinking tea.

Figure 3.43: VCD for the events corresponding to allowing, respectively disallowing, the branch leading to drinking water.

3.11.5 Communication Protocol

We have devised a metalanguage for the specification of the Communication Protocol called the *GEMOC Events Language* (GEL). Figure 3.44 shows its Abstract Syntax, specified as an Ecore metamodel.

We have developed a textual concrete syntax using Xtext [7] to enable the specification of the Communication Protocol. Listing H.1 shows an excerpt from the concrete syntax of GEL as a template (with "<" and ">" as delimiters). The full concrete syntax is shown in Appendix G.

In GEL, *Domain-Specific Events* (DSEs) implement both the "ModifierMapping" and "QueryMapping" concepts. If the referenced Execution Function is a Query, then a Feedback Policy may be specified. A Feedback Policy is composed of at least two rules, including a default one. A Feedback Rule is constituted of a Predicate on the return type of the associated Query, and of an *allowed* MoCTrigger (EventType from the MoCMapping). Since the consequences of all the rules of a policy constitute the set of data-dependent MoCTriggers, we can specify in the rules either the consistent ones or the inconsistent ones and deduce the others by getting its complement. In GEL, we have chosen to specify in the rules the MoCTriggers consistent with regards to the runtime state of the model. This syntax is more consistent with the one employed for programming languages, where conditionals are implemented through the "if…then…else" construct, and not by "if…then not…else not".

Listing 3.36 shows the Communication Protocol for fUML, specified using GEL.

Listing 3.35: Excerpt from the textual concrete syntax of GEL.

```
import <Domain metamodel>
import <Model of Concurrency Mapping>
import <Semantic Rules>

// Regular Mapping
DSE <name>(<Parameter1>, ...):
  upon <MoCTrigger from the MoCMapping>
  triggers <Execution Function from the Semantic Rules> <blocking/
   nonblocking>
end

// Mapping with a Feedback Policy
DSE <name>(<Parameter1>, ...):
  upon <MoCTrigger from the MoCMapping>
  triggers <Execution Function from the Semantic Rules> <blocking/
   nonblocking> returning <result-name>
  feedback:
    [<boolean expression using result-name>] => allow <MoCTrigger
   from the MoCMapping>
    [<boolean expression using result-name>] => allow <MoCTrigger
   from the MoCMapping>
    ...
    default => allow <MoCTrigger from the MoCMapping>
 end
end
```

Listing 3.36: The `EvaluateGuard` Domain-Specific Event (QuerryMapping) and its Feedback Policy defined in GEL.

```
import "platform:/plugin/org.gemoc.sample.fuml.model/model/fuml.
   ecore"
import "platform:/plugin/org.gemoc.sample.fuml.mocc/ECL/fuml.ecl"

DSE ExecuteActivityNode:
  upon mocc_executeNode
  triggers ActivityNode.execute
end

DSE EvaluateGuard:
 upon mocc_evaluateGuard
 triggers ActivityEdge.evaluateGuard returning result
 feedback:
  [result] => allow ActivityEdge.mocc_mayExecuteTarget
```

```
14    default => allow ActivityEdge.mocc_mayNotExecuteTarget
15   end
16 end
```

Using a model, the GEL translator is able to transform a GEL specification (Communication Protocol) into its model-level equivalent (Communication Protocol Application). The corresponding formalism is called microGEL. Its Abstract Syntax is very similar to that of GEL, except that all the language-level elements are adapted to their model-level equivalents (Execution Function into Execution Function call, MoCTrigger into MoCApplicationTrigger, etc.).

The runtime of GEL is written in Java. It takes, as input, a Scheduling Solution, and returns the corresponding MappingsApplications.

More generally, the requirements for the Communication Protocol metalanguage are the following:

- Capacity to reference elements from the MoCMapping (the MoCTriggers) and from the Semantic Rules (the Execution Functions).
- Support for arithmetic and navigation expressions on the abstract syntax concepts.
- Generator to unfold the language-level specification to the model-level (which must reference MoCApplicationTriggers and Execution Function calls).

3.11.6 Runtime

Our implementation of the Execution Engine in the GEMOC Studio is written in Java. It coordinates the runtimes for the different concerns (K3AL Executor, CCSL Solver, GEL Matcher) to drive the execution of a model conforming to a Concurrency-aware xDSML.

The GEMOC Studio also provide the possibility to define the graphical animation of the execution. This animation layer is based on the use of Sirius [14], a tool developed by Obeo which enables the definition graphical concrete syntaxes for Ecore metamodels. The implementation is based on an additional layer in the Sirius viewpoint specification, defining how to represent the model based on the evolution of its Execution Data. Then, at runtime, the graphical representation of the model is automatically updated based on the current runtime state of the model. Appendix C shows the detailed animation of the example fUML model.

[14] http://www.eclipse.org/sirius/

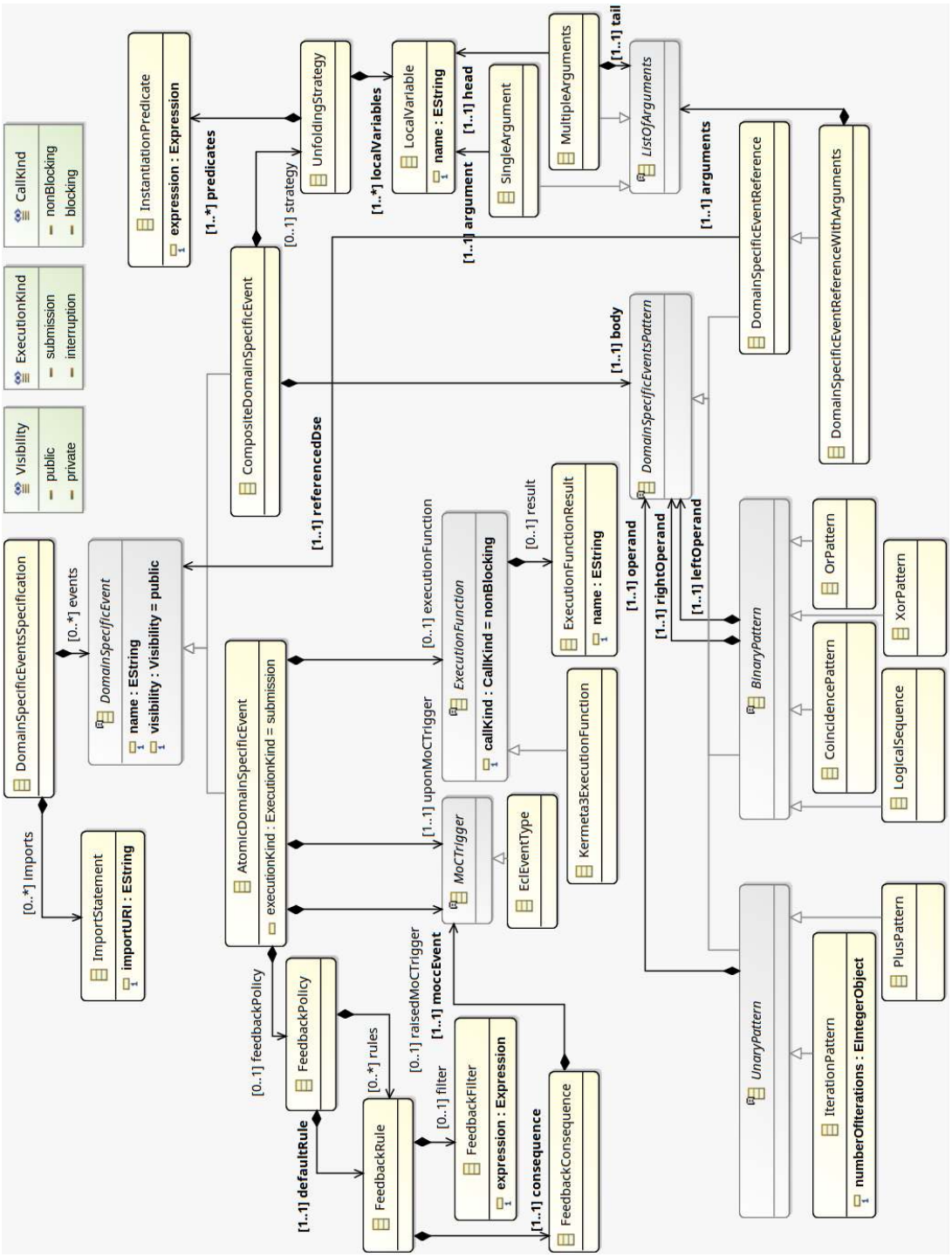

Figure 3.44: Excerpt from the metamodel representing the abstract syntax of GEL.

3.12 Conclusion

We have formalized the concurrency-aware xDSML approach developed in the context of the ANR INS GEMOC project. It is based on a separation of concerns of the operational semantics, which favors the modularity and variability of the semantics, with a focus on capturing the concurrency concerns using an adequate formalism. This formalism guarantees the correct use of a MoC by any system conforming to the xDSML, thus enabling the use of concurrency-aware analyses on the systems being designed. It also enables the refining of the xDSML for a specific execution platform, at the cost of having to respect the boundaries of the approach (*e.g.*, the MoCMapping is data-independent, Execution Function cannot call other Execution Functions, etc.).

We have improved the initial approach by identifying, motivating, illustrating and implementing a set of features which either facilitate the definition of concurrency-aware xDSMLs, or enable the specification of language constructs that could not be handled adequately in the initial approach. For instance, the reuse of Execution Functions or the addition of visibility for Mappings and the design of Composite Mappings contribute to facilitating the definition and the use of concurrency-aware xDSMLs. Features such as non-blocking Execution Function calls, the Feedback Protocol or the addition of parameters to Execution Functions and Mappings contribute to the general expressive power of the approach. We have carefully implemented these features to ensure the concurrency-awareness of the approach remains intact, retaining the modularity of the execution semantics, and making possible the independent analysis of the concurrency aspects of a model conforming to a concurrency-aware xDSML.

This approach is not the be-all of xDSML design. xDSMLs without needs for rich concurrency constructs, or high variability of its concurrent aspects, may not profit from it. Instead, it benefits xDSMLs with complex concurrent semantics, or used to design systems that are to be deployed on various execution platforms providing more or less parallel facilities. The xDSMLs can be explicitly adapted for some specific execution platform(s). It also benefits xDSMLs with Semantic Variation Points (SVPs): the use of the Event Structure MoC, which relies on partial orderings, facilitates the implementation of SVPs pertaining to the concurrency concerns of the language. The approach also benefits the design of complex systems, for which formally verifying behavioral aspects is essential for safety reasons. The systematic use of the Event Structure MoC enables performing such analyses for any model conforming to the xDSML.

Some languages cannot be captured correctly using the concurrency-aware approach. This is mainly due to the concurrency model, which is a specification *in intention* of all the possible control flows. This means that all the relevant parts of the model must be known

at compile-time. They cannot be created dynamically during the execution. Otherwise, the concurrency model would not be aware of them and would not include them in the control flow of the model.

MoCs are gaining traction in the programming community due to the complex and highly-concurrent nature of modern softwares and systems. The concurrency-aware approach eases their use through a specification at the language level. This is made possible by the domain-specificity of the language, enabling its semantics to include the systematic use of a MoC for any conforming model. This is a considerable advantage, since MoCs usually require particular training and know-how to be used correctly. The features we have presented extend the expressive power of the concurrency-aware xDSML approach, or facilitate its use, thus contributing to widening the range of xDSMLs that can benefit from the approach, ultimately improving the specification and refinement of modern software-intensive systems.

"A magical accident in the Library [...] had some time ago turned the Librarian into an orang-utan. He had since resisted all efforts to turn him back. He liked the handy long arms, the prehensile toes and the right to scratch himself in public, but most of all he liked the way all the big questions of existence had suddenly resolved themselves into a vague interest in where his next banana was coming from. It wasn't that he was unaware of the despair and nobility of the human condition. It was just that as far as he was concerned you could stuff it."

in *Sourcery*, by Terry Pratchett (1948 – 2015).

4

Tailoring Models of Concurrency to Concurrency-aware xDSMLs

Summary

We present an approach to seamlessly define and integrate new Models of Concurrency into the concurrency-aware xDSML approach presented in Chapter 3. This is done through a recursive definition of concurrency-aware xDSMLs, in which the MoC of an xDSML is a previously-defined xDSML. This allows language designers to specify the concurrency concerns of a language using the most appropriate formalism. We detail how this recursive approach impacts the specification, translation and runtime stages of the concurrency-aware approach. We also discuss its impact on the use of concurrency-aware analyses.

The contribution presented in this chapter has been published in the *2nd International Workshop on Executable Modeling* (EXE 2016) [87].

Chapter Outline

Résumé

Ce chapitre présente la solution pour un problème soulevé dans le Chapitre 3 concernant l'adéquation entre le modèle de concurrence (*Model of Concurrency – MoC*) utilisé et le *xDSML* que l'on spécifie. L'approche que nous avons décrite a été jusqu'à présent cantonnée au MoC *Event Structures*. Or, tout *MoC* n'est pas forcément idéal pour tout *xDSML*. De la même manière que certains problèmes sont plus facilement résolus à l'aide de certains langages – pousser ce raisonnement jusqu'au bout nous amenant à la programmation orientée langages (*Language-Oriented Programming – LoP*) décrite dans les chapitres précédents – différents *MoCs* correspondent à différentes façons de représenter la concurrence, et donc correspondent à différents *xDSMLs* possédant différents paradigmes de concurrence.

Dans ce chapitre, nous donnons une définition récursive de l'approche *concurrency-aware*, à travers l'utilisation d'un *concurrency-aware xDSML* en tant que *MoC* pour un autre *xDSML*. Nous insistons d'abord en détails sur l'activité de spécification du *Model of Concurrency Mapping* (*MoCMapping*). Nous identifions en effet deux étapes à cette activité : établir la correspondance entre la syntaxe abstraite du *xDSML* et la structure utilisée par le *MoC* ; et définir un ordre partiel symbolique entre les différents stimuli (*MoCTriggers*) de cette structure. Or un tel ordre partiel existe déjà entre les *Mappings* d'un concurrency-aware *xDSML*, qui peuvent donc être utilisés comme les stimuli d'un *MoC* par un nouveau concurrency-aware *xDSML*. Il ne reste alors plus qu'à définir la première étape, à savoir la correspondance entre la syntaxe abstraite de ce nouveau *xDSML*, et celle du *xDSML* utilisé en tant que *MoC*.

Pour établir cette correspondance, nous proposons de spécifier une transformation de modèles entre la syntaxe abstraite du *xDSML* et celle du *MoC*. Cette transformation permet, pour un modèle donné, d'obtenir son *MoCApplication*. Celui-ci est, entre autres, un modèle conforme à un *xDSML* (celui utilisé comme *MoC*), et peut donc être exécuté, mis au point et testé comme n'importe quel autre modèle. Un aspect important de cette transformation est qu'elle n'est *pas* une traduction du domaine du *xDSML* vers le domaine du *MoC*. Seuls les aspects liés à la concurrence du *xDSML* sont représentés à l'aide du *MoC*. En somme, le *MoCApplication* n'est pas sémantiquement équivalent au modèle initial, contrairement à ce qui est fait dans une approche translationnelle de la sémantique (*cf.* Chapitre 5).

Cette transformation peut être de type $1 \rightarrow n$, ce qui signifie qu'à un élément du modèle peuvent correspondre plusieurs éléments dans le *MoCApplication*. A l'exécution, cela peut poser des problèmes pour distinguer les multiples éléments du *MoCApplication* résultant de la transformation d'un élément du modèle initial. Pour pallier cela, nous proposons de spécifier ce que nous appelons les *Projections* du *xDSML* sur le *MoC*. Une *Projection*

définit en quel(s) concept(s) du *MoC* les concepts du *xDSML* sont transformés, et pour quelle raison (via un label). Cette spécification est semblable à un métamodèle de la trace de la transformation initialement définie. Les *Projections* sont utilisées dans la spécification du *Communication Protocol*, ce qui permet à la phase de traduction de cibler les éléments appropriés dans le *MoCApplication*.

La phase de traduction doit donc être modifiée en conséquence. Comme dans le chapitre précédent, la première étape consiste à déplier les *Semantic Rules* au niveau modèle, ce qui donne les *Semantic Rules Calls*. La seconde étape consiste à utiliser la transformation de modèles pour obtenir le *MoCApplication*. Elle permet aussi de générer les *Projections* de niveau modèle. Ces dernières sont utilisées dans la troisième étape, qui déplie le *Communication Protocol* pour le modèle considéré, créant ainsi son *Communication Protocol Application*.

Le moteur d'exécution doit lui aussi être modifié en conséquence. Le principal changement est que le *Solver* (qui sert à interpréter le *MoCApplication*) est le moteur d'exécution du *xDSML* utilisé comme *MoC*. Une couche d'adaptation est donc mise en place pour rendre compatible les interfaces du *Solver* et du moteur d'exécution.

Nous analysons ensuite cette approche récursive. Nous considérons d'abord la modularité de la sémantique d'exécution, principal avantage de l'approche originelle. Celle-ci est conservée puisque les aspects concurrents demeurent définis à l'aide de spécifications dédiées. Nous nous intéressons ensuite à la réalisation d'analyses sur les aspects concurrents d'un modèle. Par rapport à l'approche initiale, une structure d'événement est toujours disponible, mais elle n'est présente qu'au niveau des aspects concurrents du modèle ; et possiblement sous plusieurs niveaux de langages (par exemple si un *xDSML* est utilisé en tant que *MoC* pour un *xDSML* lui-même utilisé comme *MoC* du langage que l'on souhaite analyser). Une partie des aspects concurrents peut donc être analysée, au prix d'arriver à faire les traductions des propriétés et de leurs résultats entre le domaine du *xDSML* et le domaine du *MoC* utilisé. Un autre type d'analyse est possible puisque le *MoCApplication* est dans ce cas un modèle conforme à un *xDSML*. Tout outil ou méthodologie connu pour le *xDSML* utilisé comme *MoC* peut donc être utilisé pour analyser la totalité des aspects concurrents d'un modèle. Cette définition récursive donne aussi une structure systématique aux *MoCs*, qui n'était pas formellement identifiée par le passé car historiquement, les différents *MoCs* connus ont été développés dans des contextes très différents. Ainsi, passer d'un *MoC* à un autre peut-il avoir un sens à l'aide de cette approche. Pour finir, nous expliquons bien en quoi l'approche proposée est fondamentalement différente d'une définition translationnelle de la sémantique d'exécution : seuls les aspects concurrents de la sémantique sont exprimés à l'aide d'un autre formalisme.

Ce chapitre est illustré à l'aide de la définition de fUML en utilisant, comme *MoC*, un langage proposant la notion de *Thread*, similaire à ce qui est proposé par les langages de programmation généralistes comme Java. Nous illustrons les étapes de spécification, compilation et exécution. La définition de ce nouveau *xDSML* utilisé comme *MoC* est disponible dans l'Annexe D, tandis que la définition de fUML à l'aide de ce xDSML est montrée dans l'Annexe E. L'exécution de l'exemple d'activité fUML est détaillée dans l'Annexe F. Enfin, nous détaillons notre implémentation de cette contribution dans le GEMOC Studio.

Les travaux présentés dans ce chapitre ont été publiés dans le *2nd International Workshop on Executable Modeling* (EXE 2016) [87].

4.1 Introduction

Concurrency is particularly hard to represent using traditional programming techniques. Historically, computer languages have been designed as sequential by default. Expressing advanced concurrent situations required additional work, possibly using libraries tied to specific operating system calls. This has motivated the development of the GEMOC concurrency-aware xDSML approach we have presented in the previous chapter, which relies on an existing Model of Concurrency (MoC): Event Structures [160]. In this chapter, we argue that a single MoC cannot be appropriate, and thus easy to use, for all xDSMLs. This motivates the need to integrate additional MoCs into the approach. We detail the difficulties in defining and integrating new MoCs. We will then propose a recursive definition of concurrency-aware xDSMLs, enabling xDSMLs to be used as the MoC of other xDSMLs.

4.1.1 Different Models of Concurrency for Different Paradigms

Comparing the expressive power of General-purpose Programming Languages (GPLs) is usually done through informal claims, although some frameworks have been proposed to formalize this [42]. Still, most of them naturally lean towards certain classes of problems, if not in the concepts, syntax or semantics they propose, at least in their libraries, frameworks, community or execution platform. We argue that the same can be said for MoCs: although they generally aim at representing the concurrency aspects of a system, some of them are more adapted for some classes of problems. This can stem from their originating background (*i.e.*, the initial needs for the development of a MoC), from the concepts they propose, or from their surrounding tooling. It can also be more subjective, due to familiarity of the language designer with a particular MoC.

For instance, Petri nets [107, 71] are particularly adapted to represent the mutual access to resources, while the Actor model [65] focuses on the message exchanges between entities (with no shared state) of a system. Depending on the nature of the systems to be designed with an xDSML, or on the verifications we may want to perform on the MoCApplication of a system, using one MoC or the other may be preferred.

In "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [144], the authors find that one of the reasons why a Scala code-base mixes MoCs is because of inadequacies in the actor model. Using an inadequate model usually complicates the specification, leading to data races and deadlocks. Mixing MoCs can lead to complex interactions between them. Moreover, some MoCs enable the use of concurrency-

aware analyses, and mixing MoCs may impair that (*i.e.*, some parts of the system may not be analyzable).

Another example to consider is how, in the GEMOC Studio, ECL and MoCCML (*cf.* Subsection 3.11.4 of Chapter 3) can be used to specify the MoCMapping of a concurrency-aware xDSML. MoCCML was designed as a merge of two manners of expressing domain-agnostic constraints: CCSL expressions and relations [92] and automatas [69]. This fusion stemmed from the difficulty to express some constraints using only CCSL concepts.

More generally, using an adequate MoC for an xDSML is essential to ease its design and verification. In the current situation of the concurrency-aware xDSML approach, this can lead to the antipattern known as the Golden Hammer: "if all you have is a hammer, everything looks like a nail". Using an inadequate MoC can make its use complex, which manifests, in the concurrency-aware approach, in making the specification of the MoCMapping more complicated.

4.1.2 Illustrative Example

Illustrating the inadequacies of a MoC for a particular xDSML is made difficult by our use of the MoC through the notion of Model of Concurrency Mapping (MoCMapping). The MoC is used to represent the concurrency concerns of a system, but its systematic use by an xDSML is captured in the MoCMapping by the language designer. Therefore, designing the MoCMapping entails two merged challenges: the adequacy of the MoC to the class of problem addressed by the xDSML; and the adequacy of the MoCMapping to capture the language-level specification of the systematic use of a MoC.

We will illustrate this issue on an example fUML Activity, by considering its corresponding Event Structure. By showing the inadequacy of this Event Structure (relative to other possibilities) to represent the concurrency concerns of the example Activity, we infer that this inadequacy is also present for the language-level specification (MoCMapping).

We consider the fUML Activity shown on Figure 4.1.

Figure 4.2 shows the corresponding *simplified* Event Structure. Two main simplifications have been applied on this figure:

- The evaluation of the guards has been regrouped as one event, whereas they are three distinct events.
- The subtleties of representing the consequences of the guard evaluations have been simplified.

The first one complicates the Event Structure in that all three guards may be evaluated in any order, including in parallel, so it creates a lot of possible scenarios (especially consid-

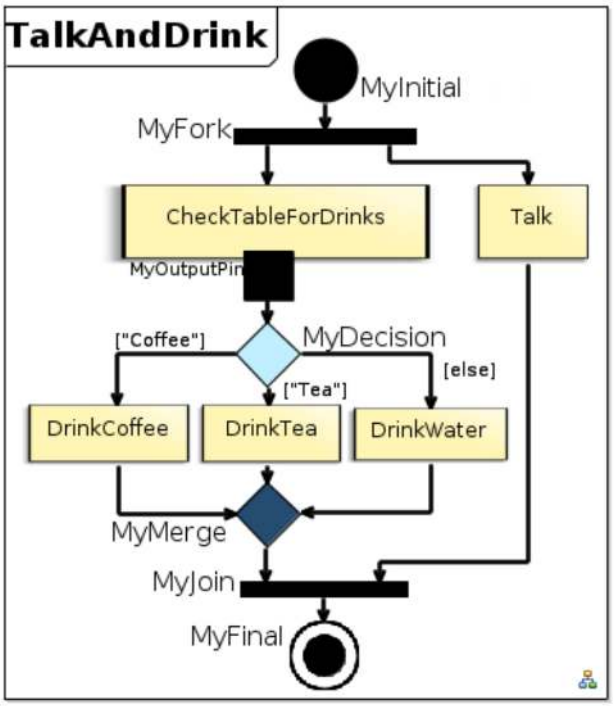

Figure 4.1: Example fUML activity where we want to drink something from the table while talking.

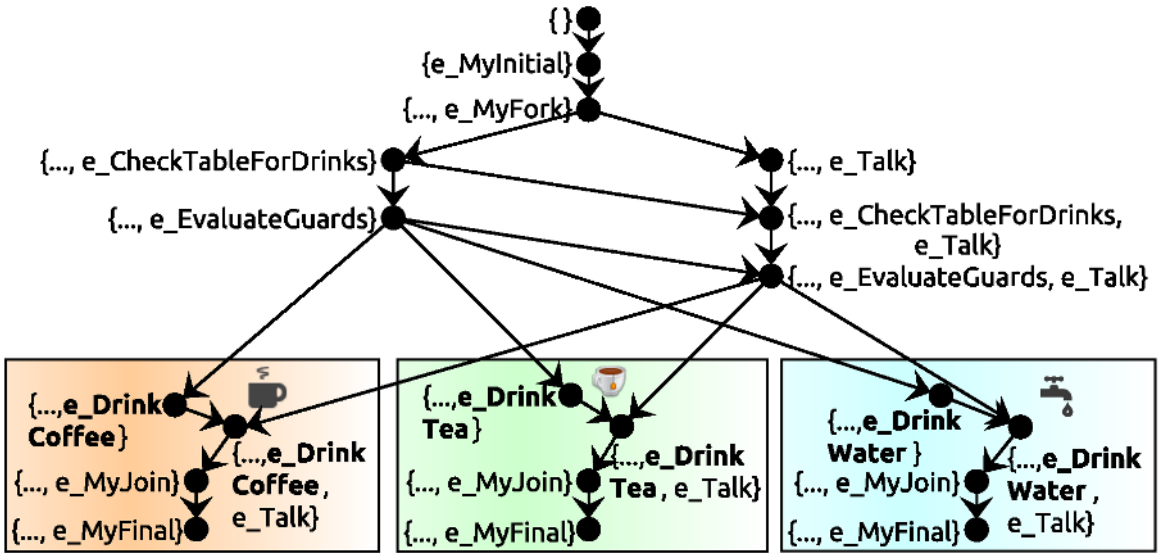

Figure 4.2: Event Structure for the fUML Activity from Figure 4.1.

ering there is another branch of the ForkNode that is executed concurrently). Appendix A paints this in greater details.

The second one is the focus of Figure 4.3, which shows a close-up on the detailed Event Structure. For each guard, we capture the consequence in terms of control flow in the "may" and "may not" events (*e.g.*,"e_mayDrinkCoffee", "e_mayNotDrinkCoffee", etc.). Each of these disjunctions must be realized based on the Feedback Protocol of the language (*cf.* Section 3.6 of Chapter 3). Afterwards, if several paths are available, then an arbitrary choice is made (with the default choice – "Water" in our case – being selected only if it is the only possible choice).

Capturing such requirements in an Event Structure is complex: there are a lot of events and specifying the right partial ordering between them is subtle due to the numerous concurrent situations. Moreover, its representation is also difficult since any concurrent situation usually leads to an exponential number of situations, *e.g.*, we cannot represent all the possible orders of evaluation of the guards while including the possible concurrency with the steps related to capturing the consequence of the evaluation of each individual guard, meanwhile concurrently executing of the other branch(es) of the ForkNode.

Instead, we propose to rely on a MoC providing the concept of *Thread*, a classical concurrency concept *inspired* from the kernel-level thread notion in Operating Systems. As mentioned in Chaper 2, the mapping between conceptual threads (also called lightweight threads, green threads, etc.) and kernel thread is realized by the underlying implemen-

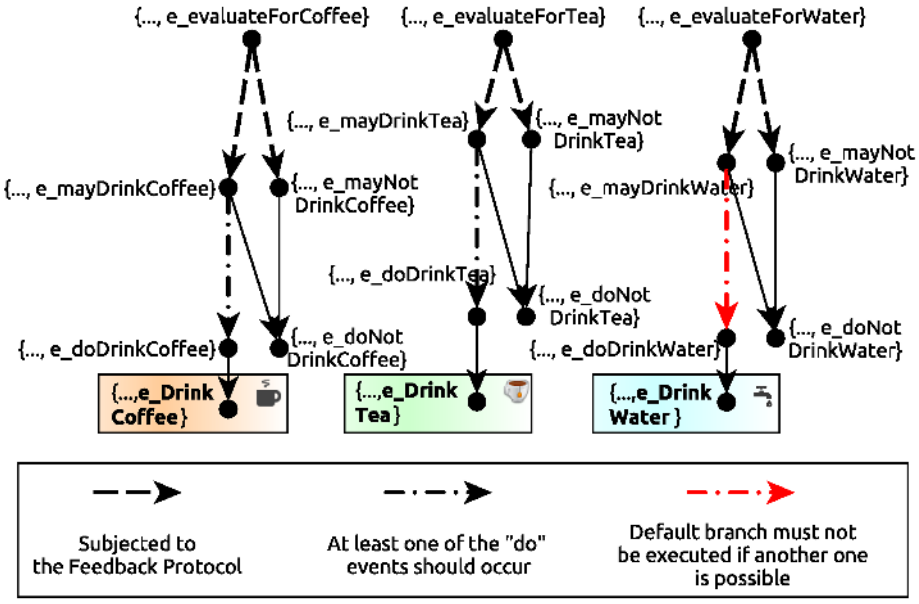

Figure 4.3: Close-up on the Event Structure.

tations. For instance, in Java, it is the JVM that dictates how Java threads are mapped to system-level threads. In the case of Oracle's HotSpot, the mapping is 1:1[1]. In other programming languages, threads are only use as a conceptual entity for a sequence of computations, and not mapped onto their own kernel thread.

A Thread is usually supplied with a list of statements (or instructions) to execute. Threads may be coordinated cooperatively, that is each Thread may relinquish control at some point. Figure 4.4 shows the use of such a MoC for the example fUML Activity.

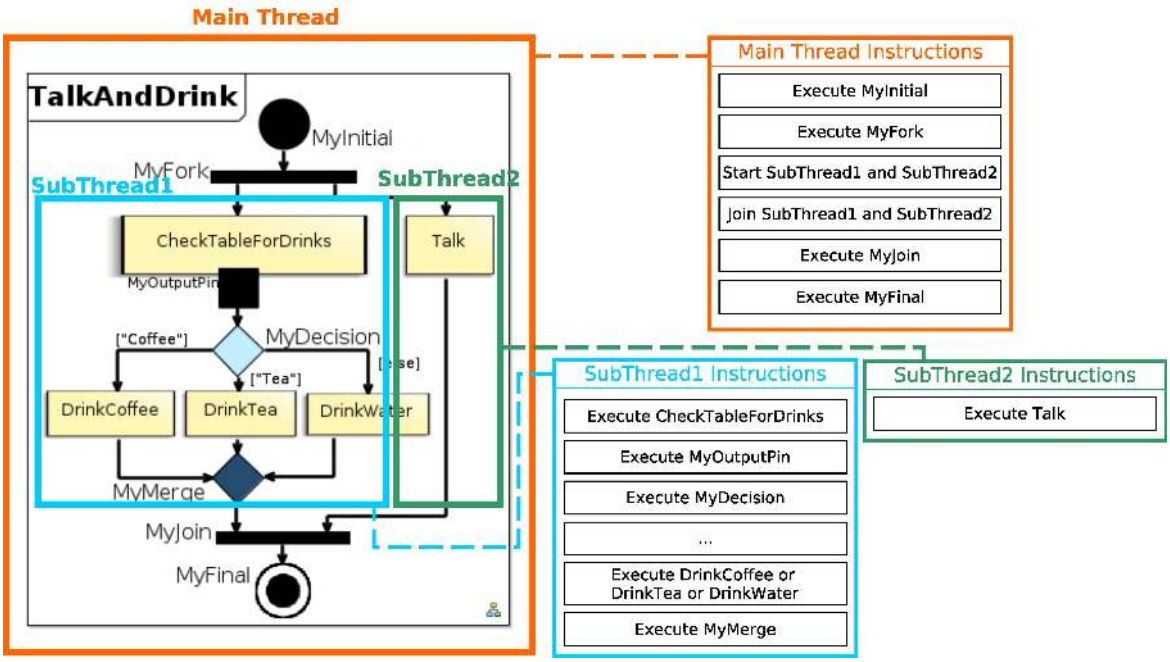

Figure 4.4: Mapping the example fUML Activity to threads.

In this particular example, we have chosen to map fUML to the notion of Threads as follows. An Activity has a main Thread. Each branch of a ForkNode/JoinNode couple is captured as a set of instruction in their own thread. The ForkNode is thus transformed into instructions corresponding to the starting of the threads of each branch. When all the Threads corresponding to branches have been fully executed, the associated JoinNode may be executed, which is captured as instructions to join (*i.e.*, wait for the completion of) a thread. For DecisionNodes, guards may be evaluated in any order, including in parallel, so using a different thread for each guard evaluation is possible. We can also simplify this aspect by executing them in any arbitrary order since it does not change the outcome.

[1]http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline

Using this thread-based MoC for fUML is more adequate than Event Structures, due to its closeness with the specification [116] and reference implementation[2]. It is also more practical to represent graphically, as all the possible interleavings between concurrent threads are not represented explicitly.

4.1.3 Integrating additional Models of Concurrency

Integrating new MoCs into the approach is complex. It requires integrating the metalanguage corresponding to the MoC. Models conforming to this metalanguage can then be used as the MoCApplication for a program conforming to an xDSML. It also requires specifying and integrating the metalanguage for the specification of the MoCMapping, as well as its translator to unfold the MoCMapping for a particular model. Finally, the runtime of the MoC must also be provided so that at runtime, the MoCApplication can be executed and interpreted by the rest of the executable model's specification.

For each MoC, the two associated metalanguages must be tooled, and their specifications and runtimes integrated with the rest of the concurrency-aware approach. Moreover, MoCs are traditionally only used at the program level, thanks to language constructs or libraries made available by the host language. The metalanguage to specify the MoCMapping is thus often not pre-existing, requiring significant efforts for its specification, development and tooling.

Additionally, there are several manners to connect a MoC (and, by extension, a MoCMapping), to the rest of the approach. For instance, for the Event Structures MoC, the connection is made by relying on the occurrences of the events. For Petri nets, one would naturally rely on the firing of transitions between places and transitions of a net. But nothing hinders us from relying instead on the entering or leaving of a place, and from interpreting these as the stimuli used by the rest of the execution of a model. Thus, identifying, for a MoC, which of its constituents' behavior will be used as the MoCTriggers is also part of how a MoC is exploited by the concurrency-aware approach.

4.2 Introducing a Recursive Definition of Concurrency-aware xDSMLs

We propose another approach to enable the use of additional MoCs. It relies on considering previously-defined concurrency-aware xDSMLs as MoCs for the design of other concurrency-aware xDSMLs.

[2]https://github.com/ModelDriven/fUML-Reference-Implementation

4.2.1 Overview of the Recursive Approach

The systematic use of a MoC by an xDSML is specified by the MoCMapping. This specification is made of two aspects. First, there is a mapping from the abstract syntax of the language to the structure used by the MoC. For instance, for an EventType Structure (language-level specification for the Event Structures MoC), it consists in defining EventTypes in the context of the concepts of the abstract syntax. The second aspect is in defining the symbolic partial ordering between the MoCTriggers (*i.e.*, in an EventType Structure, by specifying symbolic constraints between the EventTypes).

When considering a concurrency-aware xDSML, there is already a symbolic partial ordering defined between the Mappings (indirectly, as it is defined on the underlying MoCMappings). We propose to use the Mappings of a concurrency-aware xDSML as the MoCTriggers for another xDSML. This effectively allows us to reuse the symbolic partial ordering already defined for the first concurrency-aware xDSML between its Mappings. Mapping the abstract syntax of an xDSML to this structure then consists in mapping the abstract syntaxes of both languages.

Ultimately, this means that the concurrency concerns of an xDSML are represented using another xDSML. The MoCApplication will thus be a model instance of that second xDSML. Besides representing the concurrency concerns in an adapted formalism, this also means that the MoCApplication can be executed, debugged and animated like any regular model conforming to a concurrency-aware xDSML.

More formally, we denote as:

- $\mathscr{L}_{\text{Domain}}$ the concurrency-aware xDSML we are specifying;
- $\mathscr{M}_{\text{Domain}}$ a model conforming to $\mathscr{L}_{\text{Domain}}$;
- $\mathscr{L}_{\text{MoC}}$ the concurrency-aware xDSML used as a Model of Concurrency; and
- $\mathscr{M}_{\text{MoC}}$ the model conforming to $\mathscr{L}_{\text{MoC}}$ and corresponding to the MoCApplication of $\mathscr{M}_{\text{Domain}}$.

In the rest of this chapter, we will describe the specifications, translation and runtime phases of the use of $\mathscr{L}_{\text{MoC}}$ as the MoC of $\mathscr{L}_{\text{Domain}}$. $\mathscr{L}_{\text{MoC}}$ is considered as already defined, which means that it has been specified either as presented in Chapter 3 or as is being proposed in this chapter. Figure 4.5 shows an overview of the approach as a metamodel.

The specifications of the abstract syntax and of the Semantic Rules are the same as described in Chapter 3. Once again, the concrete syntax(es) and the static semantics are considered as already defined appropriately.

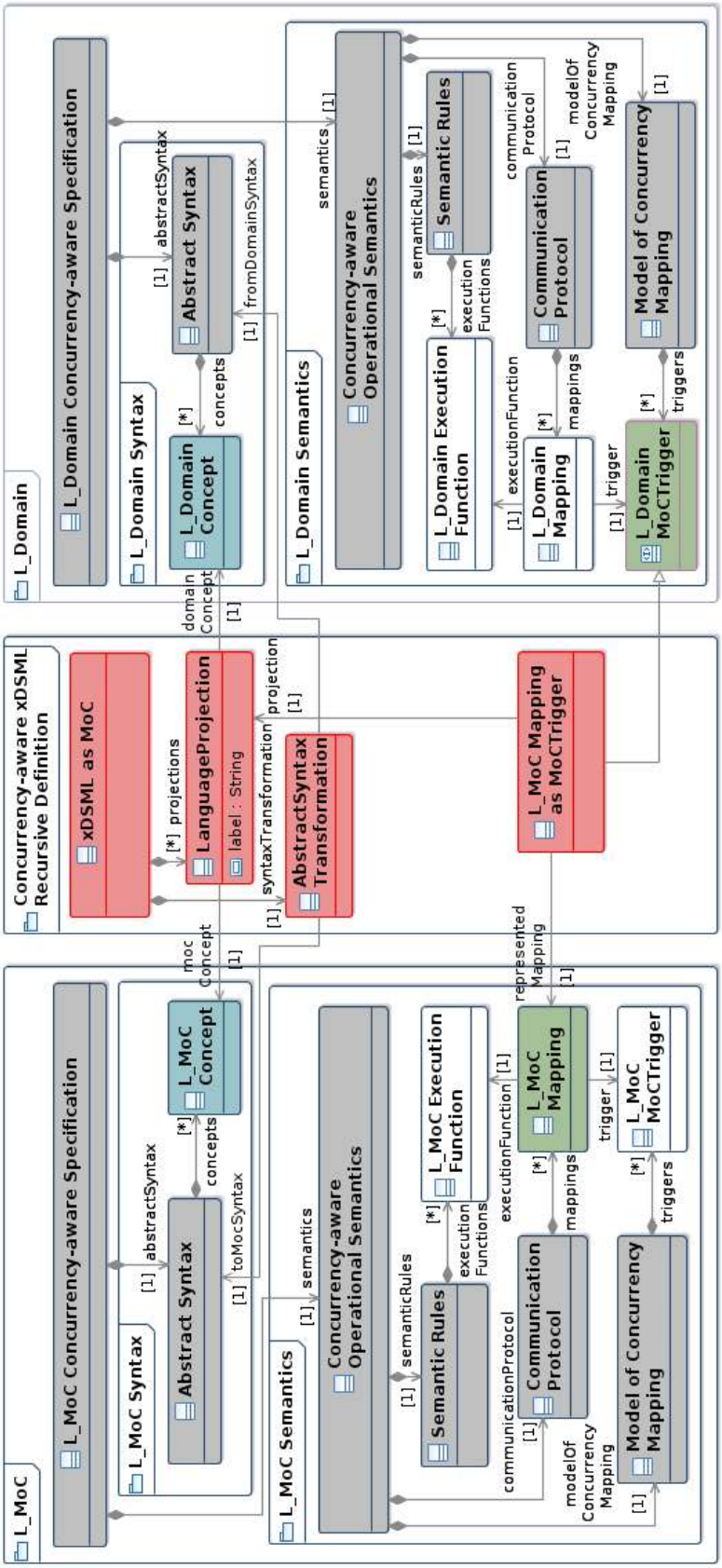

Figure 4.5: Metamodel overview of our approach for the recursive definition of concurrency-aware xDSMLs.

Our recursive definition relies on replacing the previous EventType Structure specification by the two following specifications presented in the "Concurrency-aware xDSML Recursive Definition" package of Figure 4.5. We illustrate these specifications as well as their execution on a definition of fUML using, as MoC, a concurrency-aware xDSML capturing the notions of threads with instructions.

4.2.2 Abstract Syntax Transformation

The MoCMapping is implemented by the specification named "AbstractSyntax Transformation" in the metamodel of Figure 4.5. We denote it as $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$. It specifies how the pure concurrent control flow of $\mathscr{L}_{\text{Domain}}$ is represented using $\mathscr{L}_{\text{MoC}}$. For the input model $\mathscr{M}_{\text{Domain}}$, its output is $\mathscr{M}_{\text{MoC}}$, its MoCApplication.

The Mappings of $\mathscr{L}_{\text{MoC}}$ represent the MoCTriggers of this MoC, which means that the Communication Protocol specification of $\mathscr{L}_{\text{Domain}}$ is between Mappings of $\mathscr{L}_{\text{MoC}}$ and Execution Functions of $\mathscr{L}_{\text{Domain}}$.

The correspondence between $\mathscr{L}_{\text{Domain}}$ and $\mathscr{L}_{\text{MoC}}$ is always $1 \rightarrow n$ (with $n \geq 0$). When $n = 0$, it means that the element of $\mathscr{M}_{\text{Domain}}$ has no direct impact on the control flow. When $n = 1$, the element of $\mathscr{M}_{\text{Domain}}$ is transformed into one element in $\mathscr{M}_{\text{MoC}}$. For instance, fUML nodes are generally represented by one instruction in a language based on threads and instructions (*cf.* Figure 4.4). Finally, $n > 1$ when the element of $\mathscr{M}_{\text{Domain}}$ is represented using multiple elements in $\mathscr{M}_{\text{MoC}}$, such as a ForkNode being transformed into several instructions (corresponding to starting as many threads as it has branches).

In other words, $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$ does not add new information, it merely encodes the control flow associated with the constructs of $\mathscr{L}_{\text{Domain}}$, using $\mathscr{L}_{\text{MoC}}$. The rest of the specification of $\mathscr{L}_{\text{Domain}}$ (Semantic Rules) handles the data concerns of the language.

In order to illustrate this specification on fUML, we must first consider the definition of a concurrency-aware xDSML capturing the notions of threads and their instructions. Figure 4.6 shows the Abstract Syntax and Semantic Rules of our implementation of such a language. A `ThreadSystem` is composed of `Threads` (including a main one). Each Thread has a number of `Tasks` which can be of different nature (execution, disjunction, conditional, etc.), in particular they may correspond to starting or joining other threads. Inside a Thread, Tasks are executed sequentially. Threads are concurrent by nature, so if several are running at the same time, they can execute their instructions in parallel or in some form of interleaving. Joining on another thread consists in waiting for the designated thread to have all its tasks executed. `Disjunctions` are tasks for which only one of the two operands (other Tasks) is executed. `Conditionals` are executed if all their conditions (other Tasks) have been executed previously.

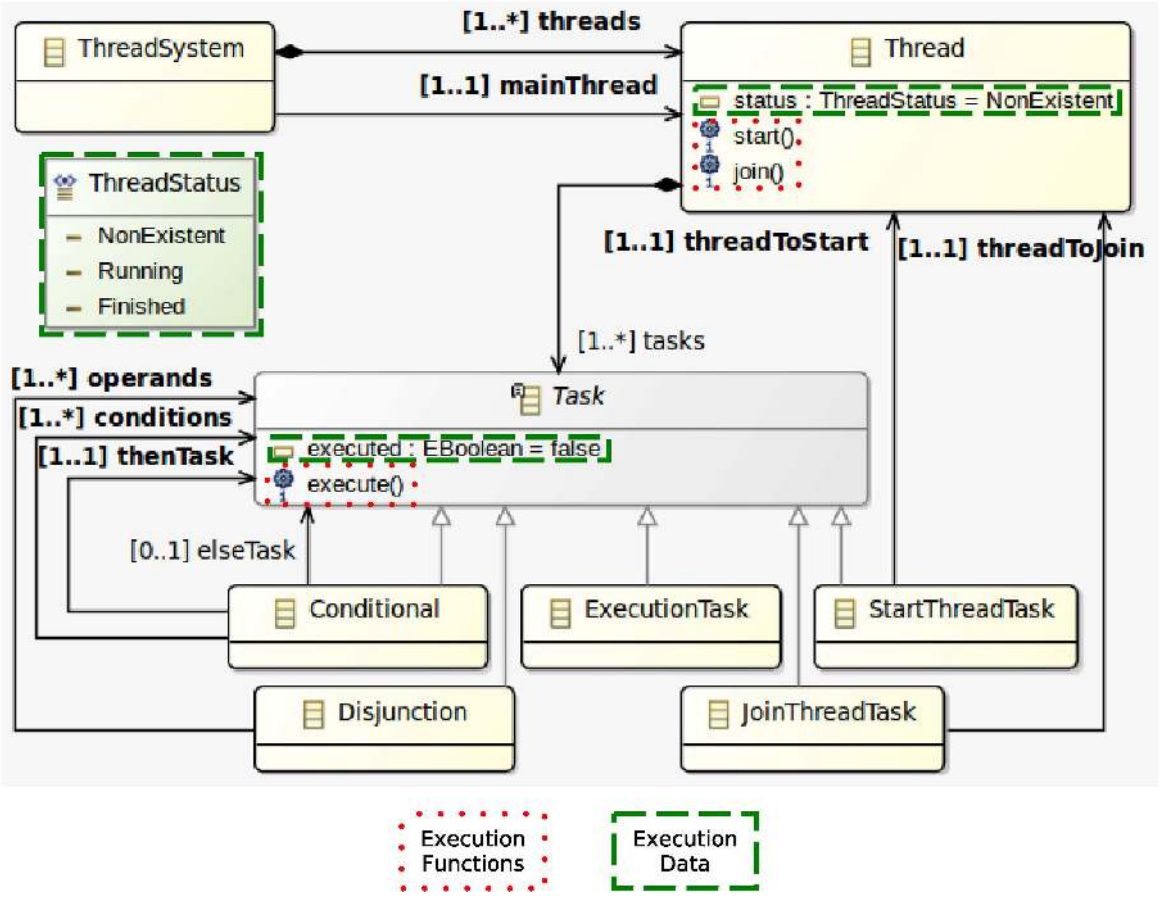

Figure 4.6: Excerpt from the Abstract Syntax and Semantic Rules of our threading language used as a MoC for fUML.

For this xDSML, the Mapping of interest in the Communication Protocol is the execution of a Task, `ExecuteTask`. Its occurrences will be used as the MoCTriggers by the Communication Protocol of fUML.

The full concurrency-aware specification of this xDSML is given in Appendix D.

For fUML, the Semantic Rules are unchanged (*cf.* Chapter 3). Our interest lies in the specification of the abstract syntax transformation, denoted as $\mathcal{T}_{\text{fUML}\rightarrow\text{Threading}}$. This transformation must produce, based on an fUML Activity, the ThreadSystem model representing its concurrency concerns. For the example Activity presented previously, this model is equivalent to the model shown in the right half of Figure 4.4, shown in a textual form using pseudo-code on Listing 4.1.

Listing 4.1: Ideal MoCApplication, based on the notion of Threads and Instructions, for the example fUML Activity.

```
1  Main Thread TalkAndDrinkActivity {
2    Execute_MyInitial;
3    StartThread SubThread1; StartThread SubThread2;
4    JoinThread SubThread1; JoinThread SubThread2;
5    Execute_MyFinal;
6  }
7
8  Thread SubThread1 {
9    Execute_CheckTableForDrinks;
10   Execute_MyOutputPin;
11   Execute_MyDecision;
12   Decision2Coffee_EvaluateGuard;
13   Disjunction { MayDrinkCoffee | MayNotDrinkCoffee };
14   Decision2Tea_EvaluateGuard;
15   Disjunction { MayDrinkTea | MayNotDrinkTea };
16   Decision2Water_EvaluateGuard;
17   Disjunction { MayDrinkWater | MayNotDrinkWater };
18   if MayDrinkCoffee and MayDrinkTea and MayDrinkWater
19     then Disjunction { Execute_DrinkCoffee | Execute_DrinkTea } end;
20   if MayDrinkCoffee and MayNotDrinkTea and MayDrinkWater
21     then Execute_DrinkCoffee end;
22   if MayNotDrinkCoffee and MayDrinkTea and MayDrinkWater
23     then Execute_DrinkTea end;
24   if MayNotDrinkCoffee and MayNotDrinkTea and MayDrinkWater
25     then Execute_DrinkWater end;
26   Execute_MyMerge;
27 }
28
29 Thread SubThread2 {
30   Execute_Talk;
31 }
```

More generally, the principles of this transformation are as follows:

- An Activity is transformed into a main Thread.
- For each pair of ForkNode/JoinNode, each branch is transformed into a Thread with Tasks corresponding to the nodes on the branch

- The ForkNode itself is transformed into a set of Tasks to start the Threads corresponding to its branches.

- The corresponding JoinNode is transformed into a set of Tasks which wait for the Threads corresponding to its branches.

- For a DecisionNode/MergeNode couple, each branch is transformed into a Task for the evaluation of its guard, and of a Disjunction between two Tasks corresponding to whether or not that branch may be executed. A set of Conditionals then describes the logics between the branches: essentially an arbitrary choice among the non-default possible ones.

- Otherwise, ActivityNodes are transformed into a single Task.

The full source code of our implementation of this transformation is available in Appendix E.

4.2.3 Using the Trace of the Abstract Syntax Transformation

Through the Abstract Syntax Transformation defined above, several concepts of $\mathscr{L}_{\text{Domain}}$ may be mapped to a same concept in $\mathscr{L}_{\text{MoC}}$, for different purposes.

In the case of fUML, edges outgoing a DecisionNode are transformed into three different instructions: one for the evaluation of their guard, and one for each possible outcome (*i.e.*, the branch is allowed, or not).

In order to ensure that the Communication Protocol of $\mathscr{L}_{\text{Domain}}$ exploits the right Mappings of $\mathscr{L}_{\text{MoC}}$, we thus need an additional specification which is based on the trace of the Abstract Syntax Transformation. This specification is denominated as the **Projections** of $\mathscr{L}_{\text{Domain}}$, denoted as $\mathscr{P}_{\text{Domain}\to\text{MoC}}$. It specifies, for a concept of $\mathscr{L}_{\text{Domain}}$, into which concept(s) of $\mathscr{L}_{\text{MoC}}$ they are transformed (through $\mathscr{T}_{\text{Domain}\to\text{MoC}}$) and with which purpose(s), using labels. This allows identifying, for instance, the different instructions corresponding to the evaluation of the guard, respectively to its different possible outcomes, resulting from the transformation of an edge outgoing a DecisionNode.

This specification is then exploited by the Communication Protocol specification of $\mathscr{L}_{\text{Domain}}$.

In the case of fUML, we denote this specification as $\mathscr{P}_{\text{fUML}\to\text{Threading}}$. Listing 4.2 shows the pseudo-code corresponding to this specification.

Listing 4.2: Pseudo-code specification of the projections of fUML onto our Threading language.

```
// Syntax:
// [Projection label]: [L_Domain concept] onto [L_MoC concept]

ProjectionForExecution: fuml.ActivityNode onto threaded.Task

ProjectionForEvaluation: fuml.ActivityEdge onto threaded.Task

ProjectionForMayExecute: fuml.ActivityEdge onto threaded.Task

ProjectionForMayNotExecute: fuml.ActivityEdge onto threaded.Task
```

Listing 4.3 shows an excerpt of the pseudo-code specification of the Communication Protocol of fUML, exploiting the Projections of fUML to ensure the right MoCTriggers from the Threading language will be used.

Listing 4.3: Excerpt from the Communication Protocol of fUML, specified using pseudo-code.

```
// Syntax:
// Mapping [mapping name]:
// upon [MoCTrigger from MoCMapping] with [Projection label from
    Projections]
// triggers [Execution Function from Semantic Rules]

Mapping ExecuteActivityNode:
  upon ExecuteTask with ProjectionForExecution
  triggers ActivityNode.execute()

Mapping EvaluateGuard:
  upon ExecuteTask with ProjectionForEvaluation
  triggers ActivityEdge.evaluateGuard()
```

4.2.4 Generation of the Model-level Specifications

The two additional specifications we have described are at the language level:

- $\mathscr{T}_{\text{DOMAIN}\to\text{MOC}}$ is a model transformation from the abstract syntax of $\mathscr{L}_{\text{DOMAIN}}$ to the abstract syntax of $\mathscr{L}_{\text{MOC}}$; it can be applied to any model conforming to $\mathscr{L}_{\text{DOMAIN}}$.
- $\mathscr{P}_{\text{DOMAIN}\to\text{MOC}}$ is a specification relating a concept from the abstract syntax of $\mathscr{L}_{\text{DOMAIN}}$ with a concept from the abstract syntax of $\mathscr{L}_{\text{MOC}}$; its model-level counterpart relates an element from $\mathscr{M}_{\text{DOMAIN}}$ with an element of $\mathscr{M}_{\text{MOC}}$.

Like in the original approach, the model-level specifications used for the execution of a model can be generated. Figure 4.7 shows an overview of how the different concerns are compiled down to the model level in the recursive approach we have described.

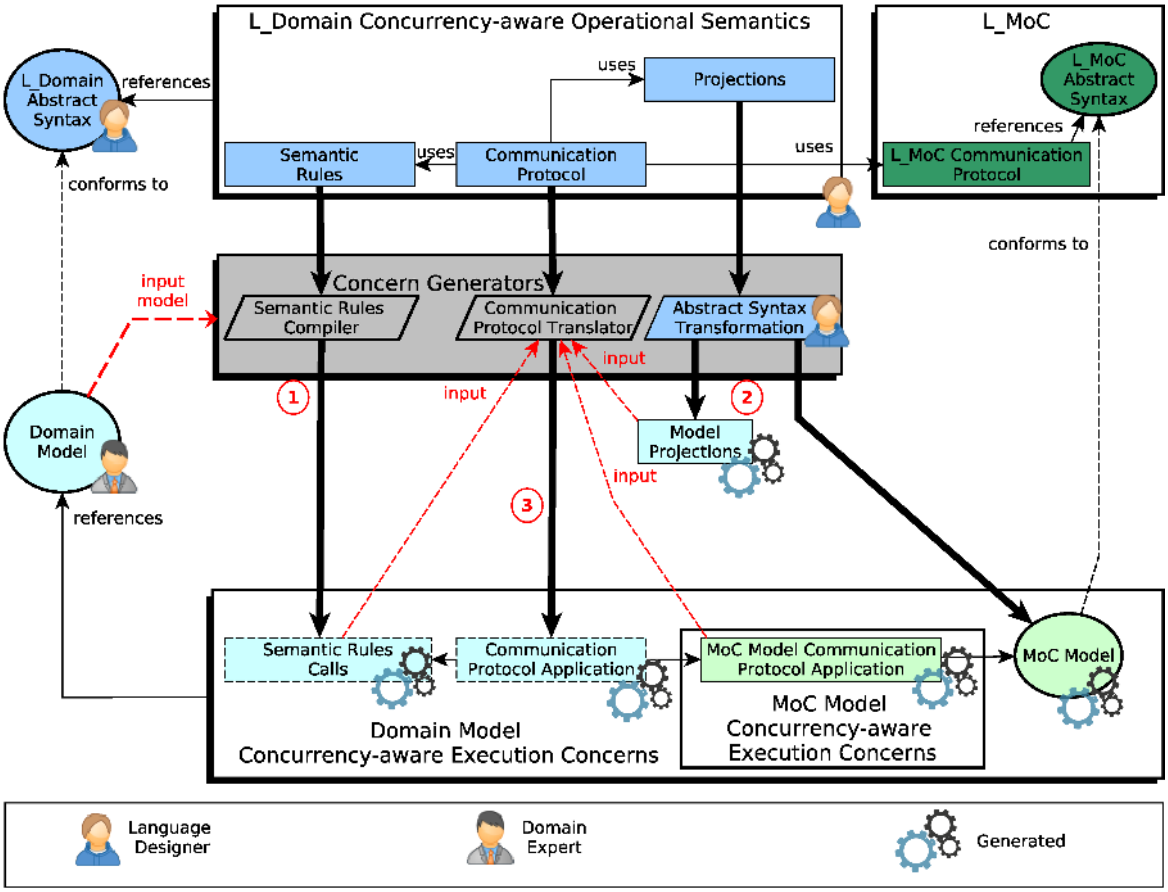

Figure 4.7: Overview of the compilation of the different concerns in our recursive approach to concurrency-aware xDSMLs.

There are three steps of generation:

1: Model + Semantic Rules → **Semantic Rules Calls**
The Semantic Rules are compiled as previously defined in Chapter 3.

2: Model + $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$ → **Model Projections** + **MoC Model (MoCApplication)**
This step corresponds to the unfolding of the MoCMapping of $\mathscr{L}_{\text{Domain}}$ to the model we want to execute. $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$ is applied using $\mathscr{M}_{\text{Domain}}$ as input, resulting in the generation of $\mathscr{M}_{\text{MoC}}$, corresponding to the MoCApplication of $\mathscr{M}_{\text{Domain}}$. Since $\mathscr{M}_{\text{MoC}}$ conforms to $\mathscr{L}_{\text{MoC}}$, its own execution concerns can be generated. In particular, its Communication Protocol Application will be used later on. During the application of $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$ (or based on the trace of its execution), the model-level projections can also be generated. We denote them as $\mathscr{P}_{\text{DomainModel}\rightarrow\text{MoCModel}}$. They map, based on the language-level projections specification, which elements of $\mathscr{M}_{\text{Domain}}$ correspond to which elements of $\mathscr{M}_{\text{MoC}}$ and with which purpose (through a label).

3: Model + Communication Protocol + Semantic Rules Calls + Model Projections + MoC Model Communication Protocol Application → **Communication Protocol Application**
This step corresponds to the generation of the Communication Protocol Application of $\mathscr{M}_{\text{Domain}}$. For each element of the model, the Communication Protocol Application maps a Semantic Rules Call to a MappingApplication of $\mathscr{M}_{\text{MoC}}$. Therefore, $\mathscr{P}_{\text{DomainModel}\rightarrow\text{MoCModel}}$ is used in order to target the right MoCApplicationTriggers of $\mathscr{M}_{\text{MoC}}$. Without it, there could be confusion when an element of $\mathscr{M}_{\text{Domain}}$ is transformed into several elements of $\mathscr{M}_{\text{MoC}}$, and thus has several potential MappingApplications available.

Let us illustrate steps 2 and 3 on our example language, fUML, using the example Activity of Figure 4.1.

The transformation, $\mathscr{T}_{\text{fUML}\rightarrow\text{Threading}}$ is used to generate the MoCApplication corresponding to our model. The resulting model, conforming to the threading language we have presented earlier, has already been illustrated, textually in Listing 4.1 and graphically in the right half of Figure 4.4. It is also used to generate the model-level projections, which are essentially parts of the trace of the application of the transformation. Listing 4.4 shows an excerpt from the model projections generated for the example Activity.

Listing 4.4: Excerpt from the Model Projections of the example fUML Activity onto the corresponding Threading model. Generated by $\mathscr{T}_{\text{fUML}\rightarrow\text{Threading}}$.

```
// Syntax:
// [label]: [M_Domain element] onto [M_MoC element]

// For all ActivityNode instances
ProjectionForExecution_MyInitial:
  MyInitial onto Execute_MyInitial
ProjectionForExecution_CheckTableForDrinks:
  CheckTableForDrinks onto Execute_CheckTableForDrinks
ProjectionForExecution_MyDecision:
  MyDecision onto Execute_MyDecision
ProjectionForExecution_DrinkCoffee:
  DrinkCoffee onto Execute_DrinkCoffee
// (...) etc.

// For all ActivityEdges instances with a guard
ProjectionForEvaluation_Decision2Coffee:
  Decision2Coffee onto Decision2Coffee_EvaluateGuard
ProjectionForMayExecute_Decision2Coffee:
  Decision2Coffee onto MayDrinkCoffee
ProjectionForMayNotExecute_Decision2Coffee:
  Decision2Coffee onto MayNotDrinkCoffee

ProjectionForEvaluation_Decision2Tea:
  Decision2Tea onto Decision2Tea_EvaluateGuard
ProjectionForMayExecute_Decision2Tea:
  Decision2Tea onto MayDrinkTea
ProjectionForMayNotExecute_Decision2Tea:
  Decision2Tea onto MayNotDrinkTea

ProjectionForEvaluation_Decision2Water:
  Decision2Water onto Decision2Water_EvaluateGuard
ProjectionForMayExecute_Decision2Water:
  Decision2Water onto MayDrinkWater
ProjectionForMayNotExecute_Decision2Water:
  Decision2Water onto MayNotDrinkWater
```

This specification is then used to generate the Communication Protocol Application for $\mathscr{M}_{\text{Domain}}$. For instance, when considering the ActivityNode *MyInitial*, of the example Activity, there is one Mapping to instantiate (*cf.* Listing 3.3), called *ExecuteActivityNode*.

Thus, there will be a corresponding MappingApplication: *ExecuteActivityNode_MyInitial*. The Mapping is specified to occur whenever the corresponding *ExecuteTask* MoCTrigger appears. However, it is possible that *MyInitial* is transformed into several different Tasks (*e.g.*, that is the case for ForkNodes), therefore there would an ambiguity as to which Task's ExecuteTask MappingApplication to use. The clause "with ProjectionForExecution" disambiguates that. We thus search, in the Model Projections, the instance of *ProjectionForExecution* for the model element *MyInitial*. We find the model projection *ProjectionForExecution_MyInitial*, which maps *MyInitial* to the Task *Execute_MyInitial*. This Task's instance of the *ExecuteTask* Mapping is thus used as the MoCApplicationTrigger for the MappingApplication of *MyInitial*.

Listing 4.5 shows an excerpt, in pseudo-code, of the resulting Communication Protocol Application.

Listing 4.5: Excerpt from the model-level Communication Protocol for our example fUML Activity, specified using pseudo-code.

```
1  // Syntax:
2  // MappingApplication [name]:
3  //   upon [MoCApplicationTrigger]
4  //   triggers [Execution Function call]
5
6  MappingApplication ExecuteActivityNode_MyInitial:
7    upon ExecuteTask_Execute_MyInitial
8    triggers MyInitial.execute()
9  // (...) etc. for every ActivityNode, the corresponding "
     ExecuteTask" through the Projection "ProjectionForExecution"
     is used to trigger the "execute()" Execution Function.
10
11 MappingApplication EvaluateGuard_Decision2Coffee:
12   upon ExecuteTask_Execute_Decision2Coffee_EvaluateGuard
13   triggers Decision2Coffee.evaluateGuard()
14 // (...) etc. for every ActivityEdge with a guard, the
     corresponding "ExecuteTask" through the Projection "
     ProjectionForEvaluation" is used to trigger the "evaluateGuard
     ()" Execution Function.
```

4.2.5 Runtime

The runtime must be changed to accommodate for the recursive definition we have presented.

Previously, we have denominated as *Solver* the runtime of the MoCApplication. In this case, the MoCApplication is $\mathscr{M}_{\text{MoC}}$, a model conforming to $\mathscr{L}_{\text{MoC}}$. Its runtime is thus an Execution Engine, itself coordinating the different runtimes for each concern of the execution of $\mathscr{M}_{\text{MoC}}$ as a concurrency-aware executable model. Thus, the Solver for $\mathscr{M}_{\text{Domain}}$ is the Execution Engine used to execute $\mathscr{M}_{\text{MoC}}$.

Performing an execution step remains similar to what was described in Chapter 3. An execution step therefore consists in:

1. retrieving the possible Scheduling Solutions from the Solver;
2. choosing an arbitrary solution among the possible ones;
3. matching the selected solution with the corresponding Execution Function calls thanks to the Communication Protocol Application; and
4. executing these calls.

In our case, a Scheduling Solution is a possible Execution Step of $\mathscr{M}_{\text{MoC}}$. Later, when the heuristic of the runtime selects one of the solutions (*e.g.*, the user through a GUI), the Solver (Execution Engine of $\mathscr{M}_{\text{MoC}}$) is notified of which step to execute, resulting in changes in the MoCApplication ($\mathscr{M}_{\text{MoC}}$). Meanwhile, the corresponding Execution Function calls of $\mathscr{M}_{\text{Domain}}$ are executed, thus concluding one step of the execution.

Overall, the main change to the runtime is that the Execution Engine must comply to the Solver interface.

The full execution of the example model is presented step-by-step in Appendix F.

4.2.6 Implementation

The approach we have described has been integrated into the Eclipse-based implementation presented in Chapter 3, the GEMOC Studio.

Specifying model transformations is a classical activity of Model-Driven Engineering (MDE) [95, 24]. GPLs can be used to write model transformations if they have access to an API able to manipulate the abstract syntax and model elements. Some languages focus on manipulating model and metamodel elements, for instance Kermeta [72] interacts well with EMOF-based models and metamodels. As mentioned in Chapter 2, the Object

Management Group (OMG)[3] has also standardized the Model to Model transformations (M2M) into QVT [114]. An example of QVT implementation is the ATLAS Transformation Language (ATL)[4] [74, 73]. Any of these means can be used to specify $\mathscr{T}_{\text{Domain}\rightarrow\text{MoC}}$. The transformation must also generate the Model Projections (*i.e.*, the trace that relates elements of $\mathscr{M}_{\text{Domain}}$ to elements of $\mathscr{M}_{\text{MoC}}$). Our implementation was made using Xtend [7] and the EMF APIs. The full source code is available in Appendix E.

The Projections can be specified using a dedicated metalanguage. Our implementation is based on the Eclipse Modeling Framework [36] and Xtext [7] (for its textual concrete syntax). Figure D.1 shows the Abstract Syntax, as an Ecore metamodel, of our implementation of this metalanguage. The language is used for both the language-level specification and the model-level specification (generated automatically by the abstract syntax transformation from $\mathscr{L}_{\text{Domain}}$ to $\mathscr{L}_{\text{MoC}}$). Its textual concrete syntax is available in Appendix H. Listing 4.6 shows the Projections of fUML specified using our metalanguage.

Finally, the metalanguage for the Communication Protocol, GEL, has been augmented to take into account our recursive approach. MoCTriggers can now consist of a Mapping (from $\mathscr{L}_{\text{MoC}}$) and of a reference to one of the projections from $\mathscr{P}_{\text{Domain}\rightarrow\text{MoC}}$. Listing 4.7 shows the Communication Protocol for our implementation of fUML.

We have also adapted the generator of the Communication Protocol to implement the proposal described previously on Figure 4.7.

Listing 4.6: The Projections of fUML onto the Threading language, specified using our dedicated metalanguage.

```
1  import "platform:/plugin/org.gemoc.sample.fuml.model/model/fuml.
       ecore" // Abstract Syntax of fUML
2  import "platform:/plugin/org.gemoc.sample.threaded.model/model/
       threaded.ecore" // Abstract Syntax of the Threading language
3
4  Projections:
5    Language Projection ProjectionForExecution:
6      fuml.ActivityNode projected onto threaded.Task
7    end
8    Language Projection ProjectionForEvaluation:
9      fuml.ActivityEdge projected onto threaded.Task
10   end
11   Language Projection ProjectionForMayExecute:
12     fuml.ActivityEdge projected onto threaded.Task
13   end
```

[3] http://www.omg.org/
[4] http://www.eclipse.org/atl/

```
14    Language Projection ProjectionForMayNotExecute:
15      fuml.ActivityEdge projected onto threaded.Task
16    end
17  end
```

Listing 4.7: The Communication Protocol of fUML.

```
1  import // Abstract Syntax of fUML
2  "platform:/plugin/org.gemoc.sample.fuml.model/model/fuml.ecore"
3  import // Communication Protocol of the Threading language
4  "platform:/plugin/org.gemoc.sample.threaded.dse/GEL/threaded.GEL"
5  import // Language Projections of fUML
6  "platform:/plugin/org.gemoc.sample.fuml.projections/projections/
       ToThreaded.projections"
7
8  DSE ExecuteActivityNode:
9    upon event ExecuteTask with ProjectionForExecution
10   triggers ActivityNode.execute blocking
11 end
12
13 DSE EvaluateGuard:
14   upon event ExecuteTask with ProjectionForEvaluation
15   triggers ActivityEdge.evaluateGuard returning result
16   feedback: // Feedback Protocol specification,
17             // more details in Chapter 3.
18     [ result ] => allow event ExecuteTask
19                           with ProjectionForMayExecute
20     default => allow event ExecuteTask
21                      with ProjectionForMayNotExecute
22   end
23 end
```

Figure 4.9 shows the MoCApplication for the example fUML Activity. It is a model conforming to the Threading xDSML we have defined, that is obtained automatically thanks to $\mathscr{T}_{\text{fUML}\rightarrow\text{Threading}}$.

Figure 4.10 shows the correspondances between the fUML Activity and the resulting Threading model.

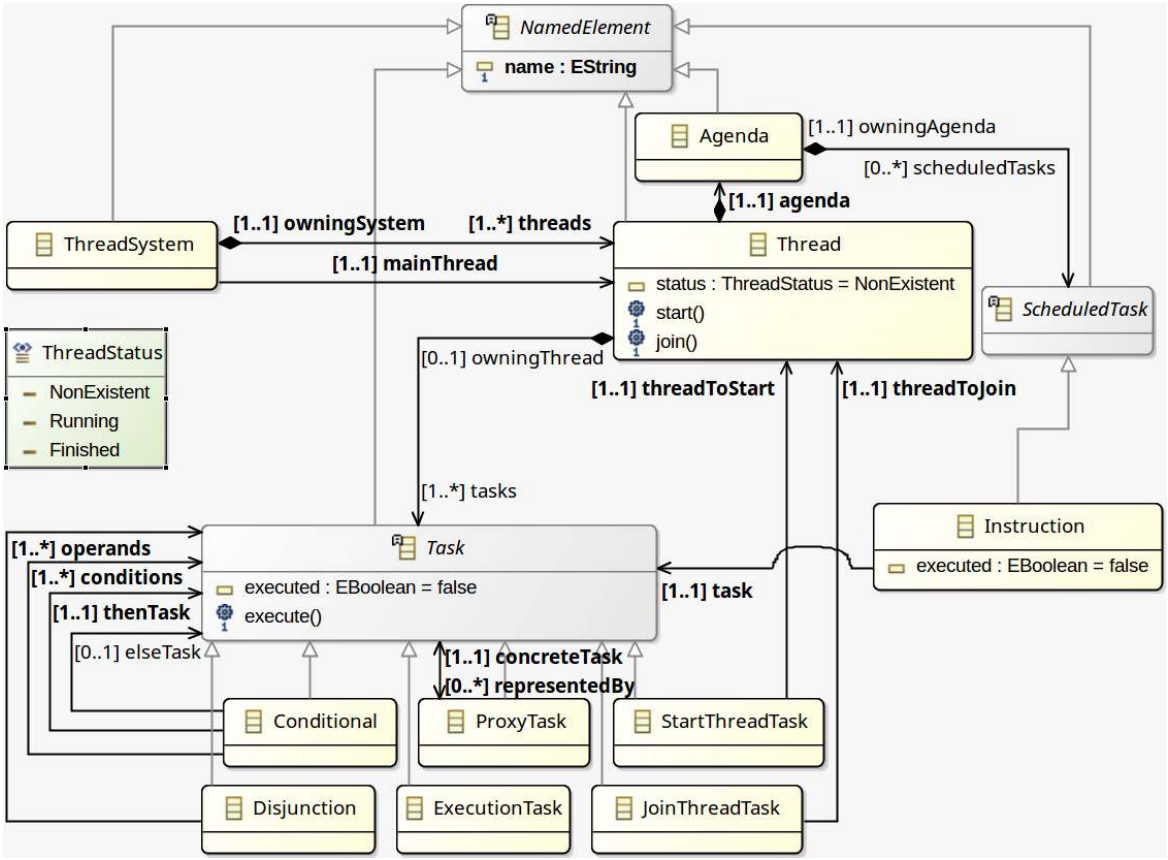

Figure 4.8: Metamodel representing the Abstract Syntax of the implementation of the Projections metalanguage.

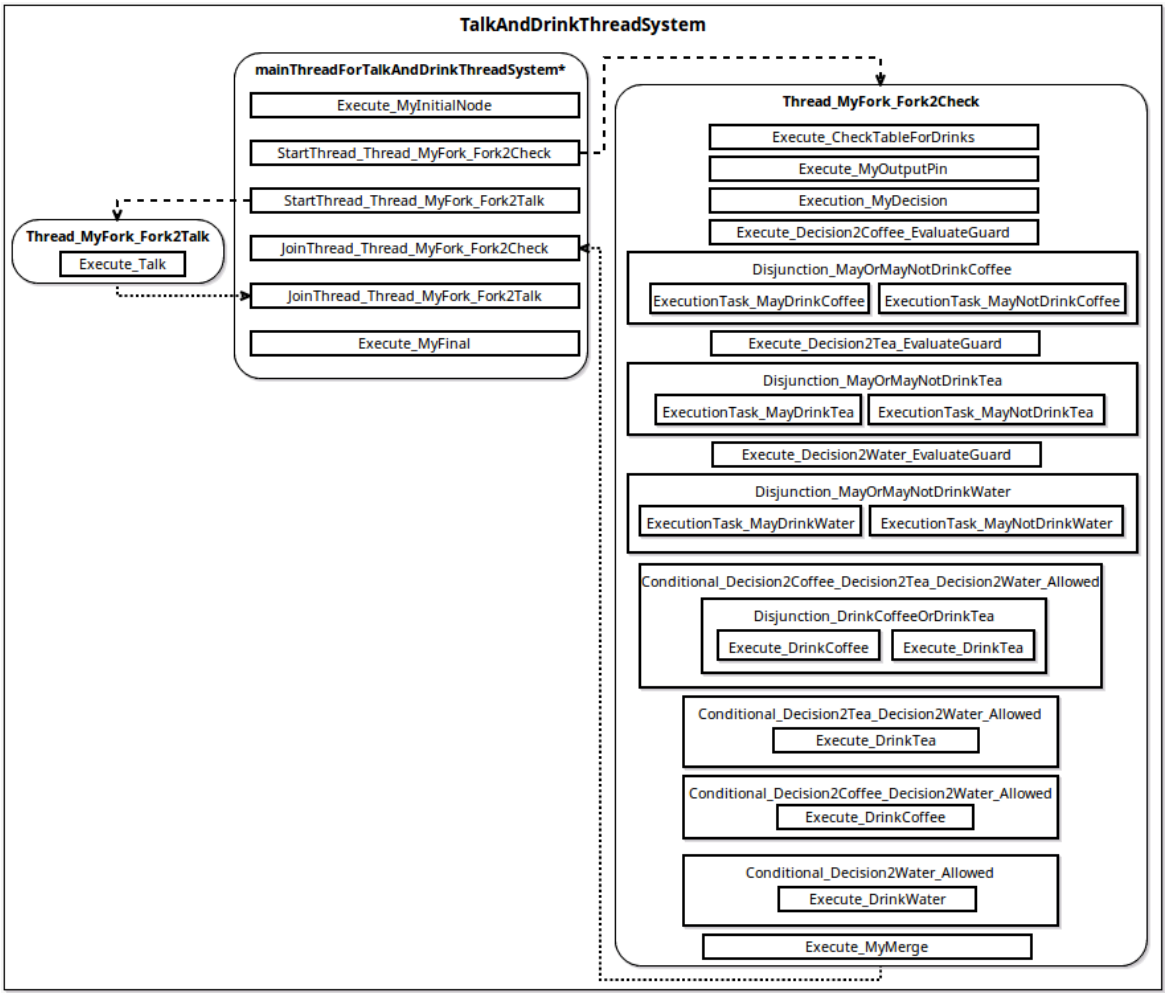

Figure 4.9: MoCApplication of the example fUML Activity, based on the Threading MoC defined as a concurrency-aware xDSML.

Figure 4.10: Correspondances between the example fUML Activity and its MoCApplication.

Figure 4.11 shows the Graphical User Interface (GUI) during the execution of the example fUML Activity. The annotated regions are as follows:

1: Graphical representation and animation of the fUML Activity, updated whenever its Execution Data evolve (*i.e.*, mostly when the tokens held by edges change).

2: Graphical representation and animation of the MoCApplication, the ThreadSystem model. Tasks in orange have been executed, while tasks in green have yet to be executed. A thread in orange has completed its execution, while a thread in green still has tasks to execute. Threads in grey have not been started yet.

3: Console used to log the different steps of execution of both models, and also used as standard output in the Execution Functions, facilitating their design and debug.

4: Set of possible Scheduling Solutions for this step. In the Threading model, there are two possibilities. Executing the next instruction of the main thread (to start the second sub-thread), executing the next instruction of the first sub-thread, or both. When matched against the Communication Protocol Application of the fUML Activity, these possibilities correspond to the three solutions visible in the "Execution Steps" view: one corresponding to "ExecuteActivityNode_MyOutputPin", one without any effect on the fUML Activity (but with some underlying effects on the Threading model), or both. When one of these solutions is selected, the corresponding Execution Functions calls are performed. For instance if the solution with both is selected, then in the fUML model, "MyOutputPin.execute()" is executed, while in its MoCApplication, both Tasks "StartThread_Thread_MyFork_Fork2Check" and "Execute_MyOutputPin" are executed.

5: Index of the active execution engines: one for the fUML Activity, and one for the Threading model.

4.3 Discussion Concerning the Recursive Approach

We discuss some aspects of the recursive concurrency-aware xDSML approach.

4.3.1 Modularity

The initial concurrency-aware xDSML approach described in Chapter 3 focuses on the separation of concerns of the execution semantics in order to make explicit the concurrency

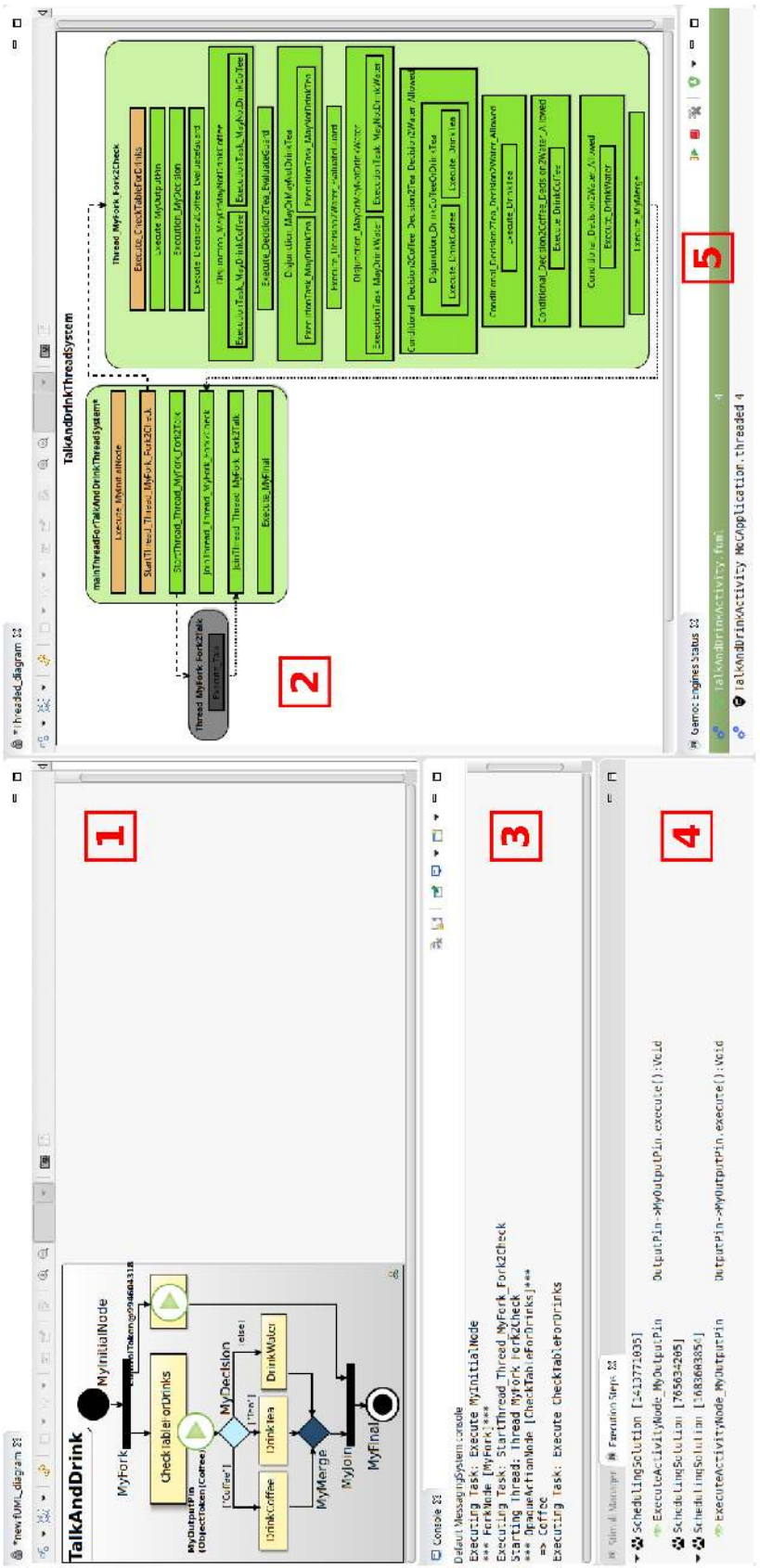

Figure 4.11: Graphical User Interface of the execution of the fUML example activity.

concerns of a language, thus facilitating its exploitation for analyses, reuse and variations. The recursive approach does not disrupt this modularity, as we have only provided the means to use other MoCs defined as concurrency-aware xDSMLs.

The MoCMapping remains a data-independent specification making explicit the systematic use of a MoC by the xDSML. In fact, our approach favors the reusability of an AS and Semantic Rules, which can be used with different MoCs, for instance to compare two MoCs for a same language in order to determine which is more appropriate. Reversely, concurrency-aware xDSML can be used as a MoC by any other xDSML.

4.3.2 Concurrency-aware Analyses

Concurrency-aware analyses can be performed on the MoCApplication of a system, depending on the MoC used. For instance, Petri nets [107, 71] are a very common formalism to specify the behavior of concurrent systems and to verify liveness or safety properties. Other xDSMLs however, may not offer such tooling or well-known properties. By enabling the use of *any* concurrency-aware xDSML as MoC, we leave into the language designer's hands the choice of using a MoC without specific properties or tooling facilitating its verification.

Still, since the concurrency-aware approach is initially based on Event Structures, there is ultimately an underlying Event Structure used for the execution. In our example, the MoCApplication of an fUML activity is a ThreadSystem model, whose MoCApplication is an Event Structure. By transitivity, we can analyze the concurrency concerns of the fUML activity through this Event Structure. However, propagating back the results of these analyses into meaningful messages for fUML may be complex. Further work could consist in providing the means to specify properties for the source model, verified on the target model, and with meaningful results being expressed for the source model [162, 163].

Overall, our approach does not hinder the use of any concurrency-aware analyses that were possible before (since we can still rely on the underlying Event Structure). It even provides an additional hook for analyses in the use of another xDSML as a MoC, possibly with specific properties or tooling available.

4.3.3 Model of Concurrency Tailored for the Concurrency Paradigm of the xDSML

By enabling the use of any concurrency-aware xDSML as a MoC, we allow language designers to use the right MoC for the xDSML being developed. This is similar to how DSLs

are used because of the dedicated abstractions they propose: some formalisms are more adapted for the specification of certain concurrency paradigms.

The use of DSLs relies on:

- being able to identify the DSL to design; and
- having the tools to specify, implement and use the DSL.

This is also the case for the use of an xDSML as MoC: it relies on identifying the fitting formalism, and on having it specified as a concurrency-aware xDSML. This may require additional work from the language designer, who must now also have an expertize in the language used as MoC, whereas previously they only needed to master the Event Structures MoC.

But for the same reasons DSLs are worth their costs, so is the recursive approach. In "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [144], mixing MoCs or using an ill-fitted MoC ultimately resulted in complex programs with deadlocks and data races, preventing the use of advanced tooling, etc. By using xDSMLs as MoCs, a practical formalism can be used for a specific xDSML's concurrency paradigm, and its use is facilitated by the possibility of executing, simulating and debugging the resulting MoCApplication just like any model conforming to a concurrency-aware xDSML.

4.3.4 Systematic Structure for Models of Concurrency

Another upside of the recursive approach is that it gives a systematic structure of the definition of a MoC. Usually, MoCs are specified informally, sometimes presented as "formalisms" (*e.g.*, Petri nets [107]), available through language constructs (*e.g.*, Erlang actors [4]) or through a framework (*e.g.*, actors in Scala/Akka [58, 55]).

Although some work has been done towards the unification of MoCs [88, 108], they mostly studied a set of MoCs, without considering the possibility to define or use new formalisms as MoCs. Using our recursive approach, the MoC used for other xDSMLs is a concurrency-aware xDSML itself. It can be used at the application level, like a regular MoC, by defining a model conforming to its syntax; and it can be used at the language level through additional specifications, like we have described in this chapter.

4.3.5 Comparison with translational semantics

The translational semantics approach consists in defining the execution semantics of a language by translating it into another well-defined language (*cf.* Chapter 2). This is usually

done through the specification of a transformation from the source language to a target language.

Our contribution bears resemblance with translational semantics in that we do define a transformation from $\mathscr{L}_{\text{DOMAIN}}$ to $\mathscr{L}_{\text{MoC}}$: $\mathscr{T}_{\text{DOMAIN}\rightarrow\text{MoC}}$. However, the *purpose* of this transformation is very different from that of translational semantics. In our approach, the source model ($\mathscr{M}_{\text{DOMAIN}}$, conforming to $\mathscr{L}_{\text{DOMAIN}}$) and the target model ($\mathscr{M}_{\text{MoC}}$, conforming to $\mathscr{L}_{\text{MoC}}$) are *not semantically equivalent.* $\mathscr{M}_{\text{MoC}}$ is only a representation of the concurrency concerns of $\mathscr{M}_{\text{DOMAIN}}$, using $\mathscr{L}_{\text{MoC}}$ as a formalism; whereas in translational semantics, the *intention* of the transformation is to produce a semantically equivalent model. The data management performed in the Semantic Rules of $\mathscr{L}_{\text{DOMAIN}}$ are never translated in terms of concepts of $\mathscr{L}_{\text{MoC}}$, and only the concurrency concerns of $\mathscr{L}_{\text{DOMAIN}}$ are transformed into $\mathscr{L}_{\text{MoC}}$.

4.4 Conclusion

In Chapter 3, we have presented the concurrency-aware xDSML approach. One of its shortcomings was that the only available Model of Concurrency was Event Structures [160]. However, this MoC is not appropriate for all xDSMLs. The adequacy of a MoC for an xDSML depends on the concurrent paradigm of its semantics, its community of users and developers, etc.

In this chapter, we have proposed a recursive definition of concurrency-aware xDSMLs. This effectively enables any previously-defined concurrency-aware xDSML to be used as the MoC for another xDSML. This recursive definition essentially relies on two specifications: $\mathscr{T}_{\text{DOMAIN}\rightarrow\text{MoC}}$, which implements the MoCMapping by defining the correspondence between the abstract syntax of the xDSML and the structure used by the MoC; and $\mathscr{P}_{\text{DOMAIN}\rightarrow\text{MoC}}$, a way to cope with the $1 \rightarrow n$ nature of the transformation. The compilation and runtime phases must also be updated to take into account these new specifications. We have implemented this contribution in the GEMOC Studio described in Chapter 3, including the new and updated metalanguages and their tools for the specification of the various concerns of the xDSML. Our example has shown how fUML can be specified using a concurrency-aware xDSML which captures the notion of Thread as its MoC, instead of Event Structures. Appendix F shows the full execution of the example fUML Activity using our new version of fUML.

The main benefit of this contribution is the possibility to rely on an appropriate formalism to specify the concurrency concerns of the xDSML. Indeed, just like computer languages are more or less adapted for some tasks, MoCs are more or less adequate to

capture the concurrency paradigm of different xDSMLs. Our contribution thus facilitates the definition and integration of new MoCs into the approach, without significant effort to make its exploitation at the language level possible (*i.e.*, the language-level metalanguage comes for free). This also opens up concurrency-aware xDSMLs to the use of other verification tools and techniques to formally ensure behavioral concurrent properties of the conforming systems. The xDSML used as MoC may be an already well-known formalism, in which case existing tools and methodologies may be used seamlessly. Further research work could consist in implementing, as concurrency-aware xDSMLs, well-known Models of Concurrency in order to reuse their properties, tools and methodologies. Another possibility is to rely on the underlying Event Structure used for the execution, but this requires additional translations of the properties and their results [163, 162].

"Apes had it worked out. No ape would philosophize, 'The mountain is and is not.' They would think, 'The banana is. I will eat the banana. There is no banana. I want another banana.'"

in *Unseen Academicals*, by Terry Pratchett (1948 – 2015).

5

Translational Semantics of Concurrency-aware xDSMLs

Summary

We propose an approach to specify the semantics of concurrency-aware xDSMLs in a translational manner, based on an existing concurrency-aware xDSML. We explain how to implement the mappings between both languages to ensure a correct definition of the newly-created xDSML.

Chapter Outline

Résumé

Ce chapitre propose une alternative pour la définition de la sémantique d'exécution d'un *concurrency-aware xDSML*. L'approche proposée dans le Chapitre 3 s'appuie sur une définition *opérationnelle* de la sémantique, tandis que dans le Chapitre 4, nous avons proposé d'utiliser un *concurrency-aware xDSML* en tant que modèle de concurrence (*Model of Concurrency – MoC*). Les aspects concurrents étaient donc définis à l'aide d'une translation vers un autre *concurrency-aware xDSML*. Nous nous attachons dans ce chapitre à donner une définition *translationnelle* de la sémantique d'exécution, c'est-à-dire *entièrement* en termes de la sémantique d'un autre *concurrency-aware xDSML*.

De manière générale, la sémantique translationnelle (ou dénotationnelle, dans le cas où l'on s'appuie sur des constructions mathématiques) consiste à définir la sémantique d'un langage en le traduisant vers un autre langage dont la sémantique est déjà définie. La première étape consiste donc à spécifier une transformation allant de la syntaxe abstraite du langage en cours de définition, vers la syntaxe abstraite du langage que nous allons exploiter. Les deux modèles (source et cible) doivent donc être sémantiquement équivalents *par construction*, et le modèle source peut être exécuté à l'aide de la sémantique déjà définie pour le langage cible.

Cependant, certaines capacités d'exécution définies dans le Chapitre 3 reposaient sur des éléments de la sémantique propres à l'approche opérationnelle. Sans ces éléments, l'exécution d'un modèle à l'aide de la sémantique translationnelle est possible, mais pas avec toutes les fonctionnalités qui étaient proposées dans le cadre d'une sémantique opérationnelle. Nous proposons donc, dans ce chapitre, de compenser ce manque. Deux aspects doivent être considérés. D'une part, la représentation de l'exécution (graphiquement dans notre cas), qui se faisait sur la base des *Execution Data*, ne peut pas être réalisée (les données sur lesquelles s'appuierait l'animation n'étant pas définies). D'autre part, l'utilisateur pouvait influencer l'exécution en jouant, pour le moteur d'exécution, le rôle de l'heuristique en charge de choisir un pas d'exécution parmi ceux rendus possibles par la sémantique. Cela n'est, pour l'instant, pas possible dans le cas d'une sémantique translationnelle, car la notion de pas d'exécution repose sur les *Mappings* du *Communication Protocol*, qui ne sont pas définis pour le langage source dans le cadre d'une sémantique par translation.

Pour le premier point, les *Execution Data* doivent être définies pour le langage source. Leur mise à jour durant l'exécution, au lieu de se faire grâce à des fonctions d'exécution, se fait à l'aide du modèle cible. Nous définissons donc une nouvelle transformation entre la syntaxe abstraite étendue du langage cible, et la syntaxe abstraite étendue du langage source. Cette transformation doit être utilisée à chaque pas d'exécution, permettant la synchronisation des *Execution Data* du langage source avec celles du langage cible.

Pour le second point, il faut tout d'abord spécifier des *Mappings* pour le langage source. Ceux-ci n'ont pas besoin d'être reliés à des fonctions d'exécution ou des déclencheurs du *MoCMapping* comme expliqué dans l'approche initiale. A la place, leurs occurrences sont déduites à partir des occurrences des *Mappings* du langage cible à l'aide d'une transformation supplémentaire. Cette transformation spécifie la correspondance des *Mappings* du langage cible vers les *Mappings* du langage source. Cette transformation est elle aussi appelée à chaque pas d'exécution, permettant à l'heuristique du moteur d'exécution de présenter (par exemple sous forme d'interface graphique) les différentes solutions possibles pour chaque pas d'exécution.

Grâce à l'ajout de ces spécifications, l'exécution d'un *xDSML* avec sémantique translationnelle est globalement équivalente, pour l'utilisateur final, à ce qui est réalisé avec une sémantique opérationnelle. Pour le concepteur de langages, utiliser la sémantique translationnelle peut être un gain de temps et d'efforts non négligeable. Spécifier des transformations de modèle est une activité classique en Ingénierie Dirigée par les Modèles, et de nombreux méta-langages peuvent être utilisés pour spécifier les transformations que nous avons décrites. Le principal inconvénient de cette approche concerne la vérification de propriétés comportementales des systèmes. En effet, le *MoCApplication* peut normalement être l'objet d'analyse des aspects concurrents du système qu'il représente. Ici, le seul *MoCApplication* qui existe est lié au modèle cible (obtenu à travers la première transformation). Il faudrait donc mettre en place une étape de transformation des propriétés, puis de leurs résultats, afin d'automatiser ces aspects de la vérification.

Cette approche translationnelle est illustrée à l'aide de l'exécution de machines à états hiérarchiques en utilisant la sémantique des machines à états non-hiérarchiques.

5.1 Introduction

5.1.1 Purpose

Translational semantics define the execution semantics of a language entirely in terms of a previously-defined executable language (a.k.a., target language). Also called *Denotational Semantics* when the translation is a mathematical denotation, it is among the three main approaches to the semantics of languages (alongside axiomatic semantics and operational semantics, *cf.* Chapter 2). The concurrency-aware xDSML approach, as described in Chapter 3 relies exclusively on operational semantics. In Chapter 4, we have proposed to partially rely on a translation to define the semantics, by using a concurrency-aware xDSML as the MoC of another xDSML. The semantics remained operational in the sense that only the concurrency concerns relied on the semantics of another xDSML. In this chapter, we propose to specify the full execution semantics of a concurrency-aware xDSML in a translational manner.

Translational semantics are practical in the sense that they completely reuse a previously-defined language, whose semantics and toolings are already available, tried, and polished. Moreover, its idea is straightforward, unlike axiomatic and operational semantics which require peculiar technologies, methodologies and trainings. The metalanguage(s) used to specify translational semantics can, more often than not, be GPLs, so long as the abstract syntaxes of the source and target formalisms offer an adequate means of manipulation (*e.g.*, often concretized as an API). Most GPLs provide the expressive power for writing such transformations ; in fact, a common early validation phase in GPL design is to write its compiler or interpreter using itself.

To specify the semantics of a concurrency-aware xDSML in a translational manner, we must rely on a pre-existing concurrency-aware xDSML, whose own semantics has been specified in an operational or translational manner. We will first define the semantics of an xDSML in a translational manner, and then detail how to make these semantics concurrency-aware, so as to benefit from the advantages of the approach presented in Chapter 3. In particular, we will show that the overhead (in terms of specifications and their complexity) induced by the concurrency-awareness of the approach is small, therefore making the use of translational semantics very approachable for concurrency-aware xDSMLs.

5.1.2 Starting Point

As in Chapter 3, we assume that any Abstract Syntax (AS), Concrete Syntax and Static Semantics issues have been resolved beforehand. Our only interest is in specifying the execution semantics of a concurrency-aware xDSML using another concurrency-aware xDSML.

Since we rely on using a previously-defined concurrency-aware xDSML, let us denote as $\mathscr{L}_{\text{TARGET}}$ this language, while $\mathscr{L}_{\text{SOURCE}}$ is the concurrency-aware xDSML for which we want to specify the execution semantics.

5.1.3 Statecharts Example

As an example, let us consider Statecharts, an extension of the Finite State Machines (FSMs) formalism. It includes hierarchy, concurrency and broadcast communications [61]. There are many dialects of Statecharts (Harel's original Statecharts [61], UML state machine diagrams [111], IBM's Rhapsody [62], etc. [5]), with differences in notation, well-formedness and semantics [22]. Differences in the latter are the most critical ones, since they are usually revealed at execution time, possibly in rare corner cases. This hinders the communication between tools, as well as between developers. Such differences are called Semantic Variation Points (SVPs), and were presented and illustrated on fUML in Section 3.8.

Let us consider a simple example representing a basic music player, shown on Figure 5.1.

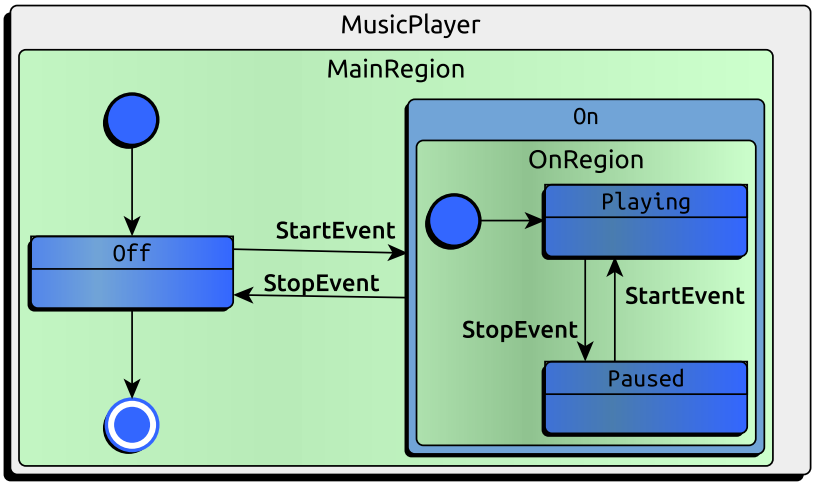

Figure 5.1: Example model of Statecharts representing a simple music player.

For this model, the semantics vary depending on how the dialect used implements the SVP called "Priorities of conflicting Transitions". When several `Transitions` are enabled by the same `Event` occurrence, how they are handled if their executions conflict (*i.e.*, their application would lead to an inconsistent model state) depends on the dialect used. In the original Harel Statecharts [63], priority is given to the Transition which is highest in the hierarchy. In UML [111], priority is given to the Transition which is lowest in the hierarchy.

For the example model, this means that when in the `States` "On" and "Playing", and the Event "StopEvent" occurs, then the Transition which is fired is either the one from "On" to "Off" (original Harel formalism) or the one from "Playing" to "Paused" (UMl [111]).

A common way to execute hierarchical Statecharts is to first *flatten* them. Removing the hierarchies simplifies the semantics by removing the ambiguities. This translation is *fully abstract* [130], in the sense that it does not alter the abstract level of the semantics: no additional details of the execution of a a Statecharts model is exposed by first flattening it. In the case of translational semantics, the SVP we are considering can be implemented by varying the strategy used to flatten the Statechart. With a first strategy corresponding to the semantics of the Harel Statecharts, the flattened Statechart is as shown on Figure 5.2. With another strategy corresponding to the semantics of the UML dialect, the resulting Statechart is as shown on Figure 5.3.

These two flattened Statecharts can be executed non-ambiguously using the semantics of Statecharts.

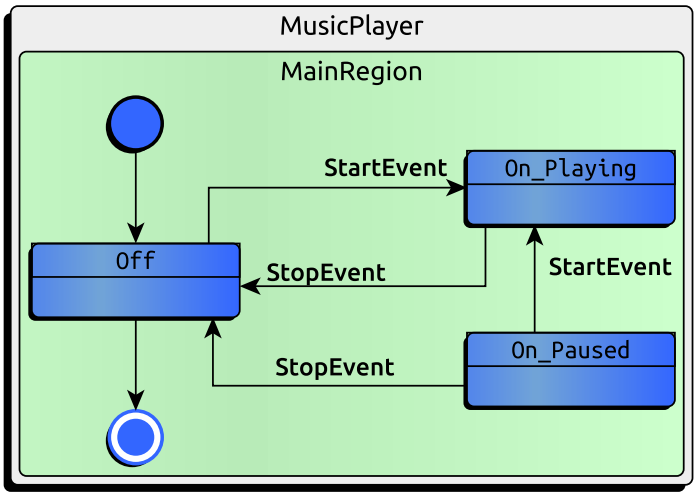

Figure 5.2: Example Statechart model flattened according to the original Harel Statecharts semantics.

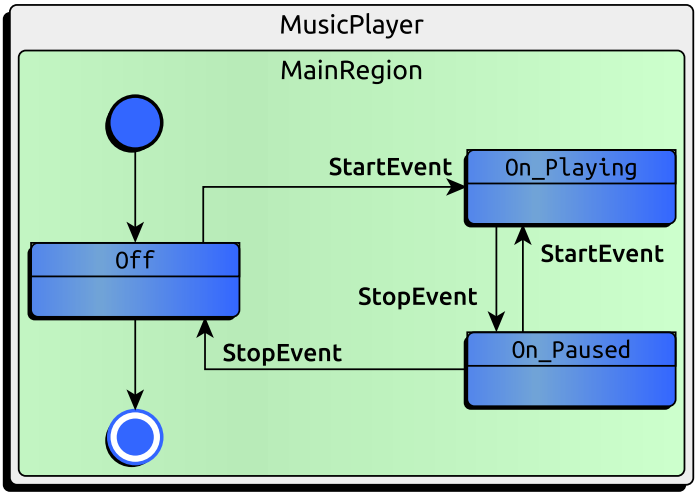

Figure 5.3: Example Statechart model flattened according to the UML semantics.

5.2 Minimal Approach to Concurrency-aware Translational Semantics

5.2.1 Main Transformation

The main artefact in a translational semantics specification is the transformation from the source language to the target language. In our case, a transformation from $\mathscr{L}_{\text{SOURCE}}$ to $\mathscr{L}_{\text{TARGET}}$, denoted as $\mathscr{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$.

This transformation specifies how any model conforming to the abstract syntax of $\mathscr{L}_{\text{SOURCE}}$ (denoted as $\mathscr{M}_{\text{SOURCE}}$) is transformed into a model conforming to the abstract syntax of $\mathscr{L}_{\text{TARGET}}$ (denoted as $\mathscr{M}_{\text{TARGET}}$). $\mathscr{M}_{\text{SOURCE}}$ and $\mathscr{M}_{\text{TARGET}}$ are thus semantically equivalent, since the application of the execution semantics of $\mathscr{L}_{\text{SOURCE}}$ to $\mathscr{M}_{\text{SOURCE}}$ is precisely $\mathscr{M}_{\text{TARGET}}$ itself.

Figure 5.4 sums up the architecture of using translational semantics for concurrency-aware xDSMLs.

Using this approach, $\mathscr{L}_{\text{SOURCE}}$ is only constituted of an abstract syntax and of $\mathscr{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$. Any model $\mathscr{M}_{\text{SOURCE}}$ can be executed thanks to the semantics of $\mathscr{L}_{\text{TARGET}}$.

5.2.2 Shortcomings

The approach so far is quite straightforward and allows the definition of executable models. But the execution of $\mathscr{M}_{\text{SOURCE}}$ is not up to par with the execution of a model conforming to a concurrency-aware xDSMLs with operationally-specified semantics.

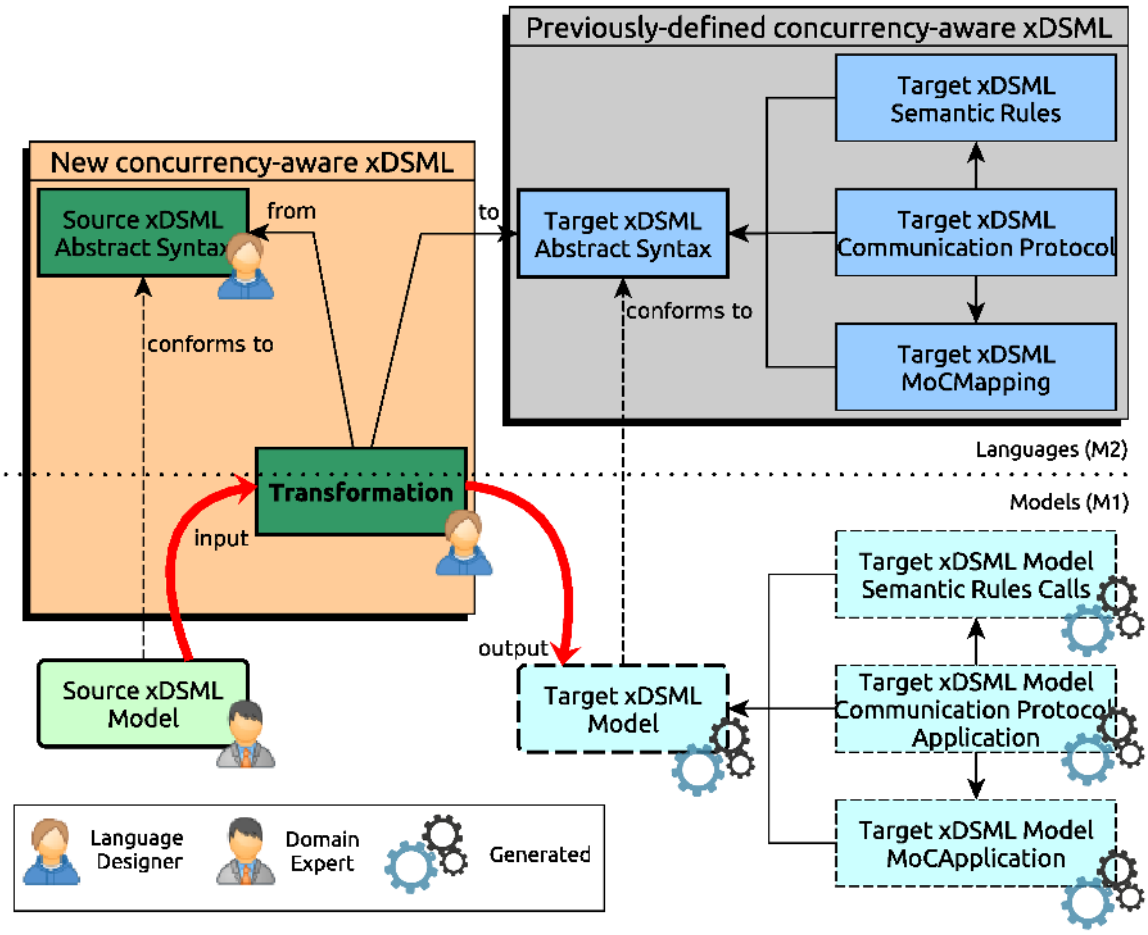

Figure 5.4: Global view of the use of translational semantics for the specification of concurrency-aware xDSMLs.

First, the notion of "runtime state" for $\mathscr{M}_{\text{SOURCE}}$ does not exist, since we have not defined any Execution Data for $\mathscr{L}_{\text{SOURCE}}$. Thus, we cannot represent it in any way (*e.g.*, graphically as shown in Appendix C). In order to get a glimpse of the current runtime state of $\mathscr{M}_{\text{SOURCE}}$ during its execution, one may observe $\mathscr{M}_{\text{TARGET}}$ and deduce, based on $\mathscr{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$, to which abstract state of $\mathscr{M}_{\text{SOURCE}}$ it corresponds to. We propose to augment the approach with an additional specification so as to explicitly capture the runtime state of $\mathscr{L}_{\text{SOURCE}}$, and maintain it consistent with the runtime state of $\mathscr{M}_{\text{TARGET}}$ during the execution, making the animation of $\mathscr{M}_{\text{SOURCE}}$ possible.

Moreover, as explained in Chapter 3, the execution of a model is driven by a heuristic of the runtime to make arbitrary choices among the possible Scheduling Solutions. In particular this heuristic can be implemented as a Graphical User Interface presenting the occurring Mappings and associated Execution Function Calls, allowing the end-user to

finely drive the execution. In the translational semantics approach we have defined so far, the heuristic is based on the Mappings of the Communication Protocol of $\mathscr{L}_{\text{TARGET}}$. The end user, supposedly familiar with $\mathscr{L}_{\text{SOURCE}}$, may not be familiar with $\mathscr{L}_{\text{TARGET}}$. Therefore, performing arbitrary choices among the Mappings of $\mathscr{L}_{\text{TARGET}}$ is not adapted for the end user. We propose to extend the approach with additional specifications to make the heuristic of the execution engine be based on the Mappings $\mathscr{L}_{\text{SOURCE}}$ instead of those of $\mathscr{L}_{\text{TARGET}}$.

In short, we are able to execute models using the translational semantics, but we don't have any meaningful (*i.e.*, belonging to the domain represented by $\mathscr{L}_{\text{SOURCE}}$) feedback or control on the execution. We propose to make this possible thanks to a few additional specifications.

5.3 Enhancing the Concurrency-aware Translational Semantics Specification

5.3.1 Animation of the xDSML

In order to ensure that we can represent the animation of $\mathscr{M}_{\text{SOURCE}}$, we first need to define the Execution Data of $\mathscr{L}_{\text{SOURCE}}$. We have introduced the notion of Execution Data (ED) in Subsection 3.2.1. In short, they are the attributes and references which, weaved into the Abstract Syntax of the language, represent the runtime state of a model during its execution. The Animation Data (*cf.* Subsection 3.3.1), used to represent the runtime state of a model in the animation layer, is then specified based on the Execution Data. In our case, the graphical concrete syntax is used to present to the end user the animation of the model's execution.

In Chapter 3, the Execution Data evolve when the Execution Functions are called. In the translational approach we are considering, the Execution Data evolve when the underlying model's runtime state evolves, *i.e.*, when $\mathscr{M}_{\text{SOURCE}}$ evolves. Since its semantics corresponds, by construction, to $\mathscr{M}_{\text{TARGET}}$, we need to maintain the consistency between $\mathscr{M}_{\text{SOURCE}}$ and $\mathscr{M}_{\text{TARGET}}$, such that whenever $\mathscr{M}_{\text{TARGET}}$ evolves, $\mathscr{M}_{\text{SOURCE}}$ evolves correspondingly.

To establish this consistency, we specify an additional transformation, denoted as $\mathscr{T}^{\text{AS + ED}}_{\text{TARGET}\rightarrow\text{SOURCE}}$. This transformation specifies how the extended abstract syntax (*i.e.*, abstract syntax plus Execution Data) of $\mathscr{L}_{\text{TARGET}}$ are transformed into the extended abstract syntax of $\mathscr{L}_{\text{SOURCE}}$. At runtime, $\mathscr{T}^{\text{AS + ED}}_{\text{TARGET}\rightarrow\text{SOURCE}}$ must be performed after each execution step of $\mathscr{M}_{\text{TARGET}}$. As a consequence, it updates the runtime state of $\mathscr{M}_{\text{SOURCE}}$, thus enabling the animation of the execution of $\mathscr{M}_{\text{SOURCE}}$.

$\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\to\text{SOURCE}}$ must be *consistent* with $\mathcal{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$. Passing a model through $\mathcal{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$ and then through $\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\to\text{SOURCE}}$ must yield the original model. This is precisely one characteristic of *Bidirectional Model Transformations* [66, 139, 138, 25], which capture, in a single transformation (considered *bidirectional*), two complementing transformations. In our case, it could be used to define, as one artefact, both $\mathcal{T}^{\text{AS}}_{\text{SOURCE}\to\text{TARGET}}$ and $\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\to\text{SOURCE}}$. Maintaining two transformations in coherence is also possible, albeit more prone to errors.

5.3.2 Heuristic of the Execution Engine

In order to enable the heuristic of the runtime to be based on $\mathscr{L}_{\text{SOURCE}}$, we must add three specifications.

First, we must add the Mappings of $\mathscr{L}_{\text{SOURCE}}$ upon which the heuristic of the runtime will be based (*i.e.*, the ones constituting the behavioral interface of the xDSML). But, contrary to how it was done in Chapter 3, we do not need to map them to a MoCTrigger or to an Execution Function. We only have to declare them with a name (and visibility and parameters if these features are implemented). These Mappings will act merely as an $\mathscr{L}_{\text{SOURCE}}$-meaningful interface on top of the Mappings of $\mathscr{L}_{\text{TARGET}}$.

Then, we must add a specification denoted as $\mathcal{T}^{\text{Mappings}}_{\text{TARGET}\to\text{SOURCE}}$ which expresses how Mappings from $\mathscr{L}_{\text{TARGET}}$ are transformed into Mappings from $\mathscr{L}_{\text{SOURCE}}$. This transformation is used for every execution step of $\mathscr{M}_{\text{TARGET}}$, retrieving the possibly occurring Mappings based on the available Scheduling Solutions, presenting them in the heuristic as Mappings of $\mathscr{L}_{\text{SOURCE}}$, and upon selection executing them as their corresponding Mappings of $\mathscr{L}_{\text{TARGET}}$.

$\mathcal{T}^{\text{Mappings}}_{\text{TARGET}\to\text{SOURCE}}$ is specified between the Mappings of $\mathscr{L}_{\text{TARGET}}$ and $\mathscr{L}_{\text{SOURCE}}$, but it is applied on the MappingApplications of $\mathscr{M}_{\text{TARGET}}$ and $\mathscr{M}_{\text{SOURCE}}$. This means that $\mathcal{T}^{\text{Mappings}}_{\text{TARGET}\to\text{SOURCE}}$ is essentially a Higher-Order-Transformation, *i.e.*, a transformation which produces another transformation.

Figure 5.5 sums up the integration of the two additional specifications, $\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\to\text{SOURCE}}$ and $\mathcal{T}^{\text{Mappings}}_{\text{TARGET}\to\text{SOURCE}}$, into the approach.

5.3.3 Application to Statecharts

Illustrating our approach on Statecharts is quite straightforward because the target xDSML is a subset of the source xDSML.

The Execution Data of Statecharts is mostly captured in the notion of "current state" of a state machine and of a composite state. The transformation from the target xDSML to the source xDSML is then as follows. When the current state is "On_Playing", then in the

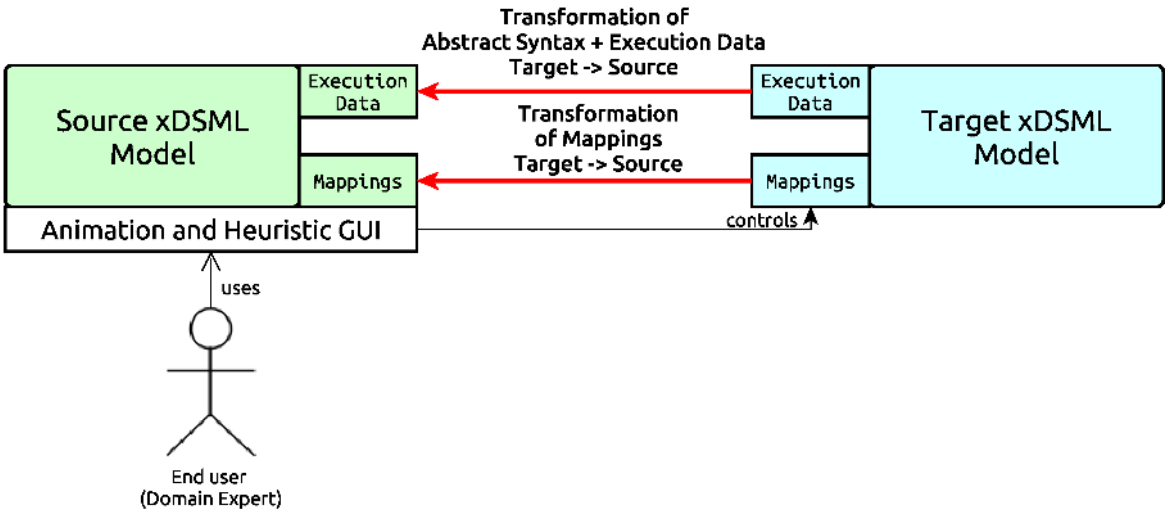

Figure 5.5: Overview of the integration of the $\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\rightarrow\text{SOURCE}}$ and $\mathcal{T}^{\text{MAPPINGS}}_{\text{TARGET}\rightarrow\text{SOURCE}}$ in the concurrency-aware xDSML approach.

original model the current state of the state machine is “On” and the current state of the “On” state is “Playing”. Respectively for “On_Paused”, it is “On” and “Paused”.

The Mapping transformation is also straightforward, since it is mostly about the firing of the transitions. The firing of a transition in the target xDSML corresponds to the firing of a transition in the source xDSML.

5.3.4 Runtime

Figure 5.6 shows an overview of the runtime as a simplified sequence diagram. Only one step of execution is represented.

The first step consists in retrieving the occurring MappingApplications for the source model. This is done by first retrieving the occurring MappingApplications in the target model, and then using the transformation resulting from $\mathcal{T}^{\text{MAPPINGS}}_{\text{TARGET}\rightarrow\text{SOURCE}}$ to map them to their corresponding source model MappingApplications. These can then be sent to the heuristic of the runtime, whose role is to select one of them arbitrarily (possibly through a graphical user interface, or through an external program using an API).

The second step consists in executing the selected step. The runtime of the target model reverse-matches the selected step to deduce which MappingApplications of the target model must be executed. Then, like in the runtime presented in Chapter 3, the Execution Function Calls are executed, resulting in changes in the target model.

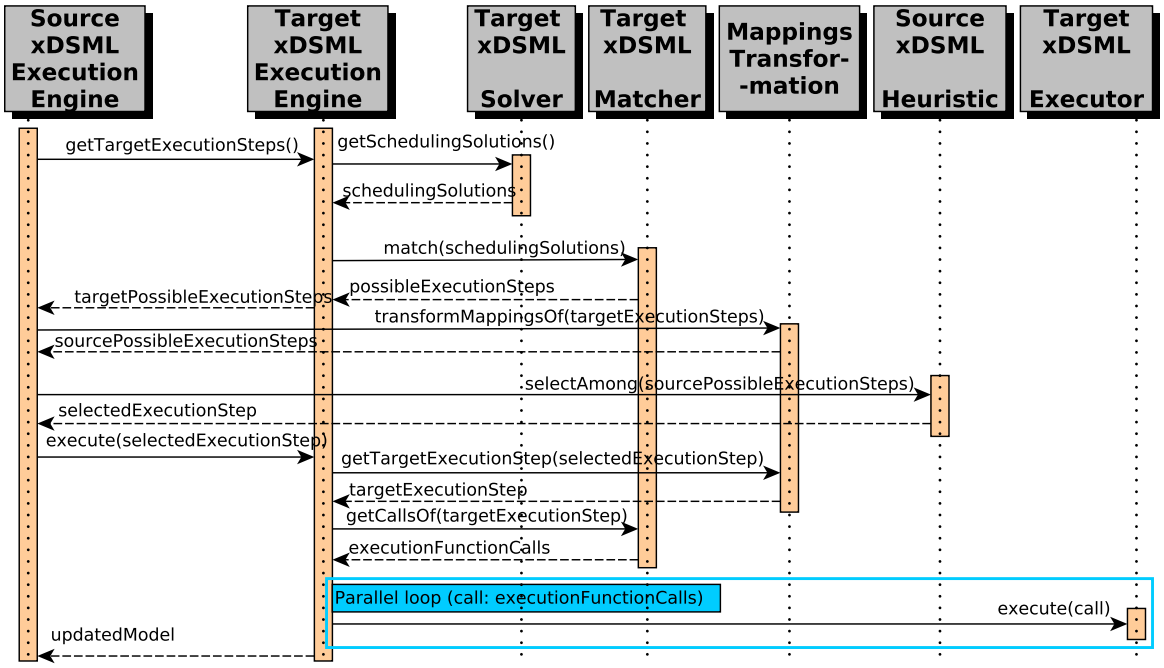

Figure 5.6: Simplified sequence diagram of the runtime using translational semantics for concurrency-aware xDSMLs.

Finally, the last step consists in communicating the updated target model so that the runtime for the source model can use the transformation resulting from $\mathcal{T}^{\text{AS + ED}}_{\text{TARGET}\rightarrow\text{SOURCE}}$ to update the source model. The animation layer can then represent to the end user the new runtime state of the model being executed.

5.4 Conclusion

We have described how translational semantics can be used to specify the execution semantics of xDSMLs. In order to benefit from all the concurrency-aware execution facilities provided by the concurrency-aware approach, we must specify additional transformations. They are mainly used to maintain the runtime state of the source model consistent with the runtime state of the target model, and to enable the heuristic of the runtime to be meaningful for the source domain, understood by the end user.

Translational semantics are practical because they rely on reusing a whole previously-defined xDSML, therefore saving the language designer most of the effort of expressing the execution semantics. However, in the context of the concurrency-aware approach, it does come with the following downside. The concurrency of a model is not directly avail-

able anymore. Indeed, the only MoCApplication available is the one for the target model (the result of the translation to the target domain) and thus it represents the concurrency concerns of the target model. Therefore, any properties validated on this MoCApplication need to be translated to the source domain if we want the end user to benefit from them. Similar to the recursive approach defined in Chapter 4, concurrency-aware translational semantics remain rooted in Event Structures: there is, ultimately, an underlying Event Structure used for the execution. By transitivity, we can analyze the concurrency concerns of a Statechart through its flattened equivalent's Event Structure. However, propagating back the results of these analyses, in a meaningful way for hierarchical Statecharts may be difficult. Further work could integrate the possibility to translate properties specified on the source model into properties of a target model, with a translation of the results for the source model [162, 163].

Compared to the contribution of Chapter 4, we have two semantically equivalent model, which means that there are two ways to analyze the source model. It can be (fully) analyzed through its target model; and its concurrency concerns can be analyzed through its target model's concurrency concerns. In the case of the recursive definition proposed previously, the result of the model transformation was *not* semantically equivalent and only represented the concurrency concerns. The data concerns were not translated to another formalism. In the approach proposed in this chapter, the whole semantics is expressed using the target xDSML.

"All we have to decide is what to do with the time that is given us."

in *The Fellowship of the Ring*, by J. R. R. Tolkien (1892 – 1973).

6

Conclusion and Perspectives

Summary

We conclude this thesis by summing up our contributions to the design, implementation and execution of concurrency-aware xDSMLs. We propose some perspectives of future work to improve on the approach.

Chapter Outline

Résumé

Ce chapitre conclut la présentation de notre travail et propose des pistes de recherche dans la continuité de ce que nous avons réalisé.

Les systèmes et logiciels modernes, hautement concurrents, et devant s'exécuter sur des environnements de plus en plus parallèles, conduisent au développement de nouveaux paradigmes de génie logiciel. Dans cette thèse, nous avons participé à l'étude du rapprochement de deux domaines de recherche : la programmation orientée languages (*Language-Oriented Programming – LOP*) et les modèles de concurrence (*Models of Concurrency – MoCs*). Nous avons détaillé comment le résultat de ce rapprochement, les langages de modélisation dédiés exécutables (*eXecutable Domain-Specific Modeling Languages – xDSMLs*) dans lesquels la concurrence est explicite et exprimée à l'aide d'un formalisme adapté (*concurrency-aware xDSMLs*), peuvent être implémentés afin de concevoir les systèmes de demain.

La force de ces langages réside dans l'utilisation systématique d'un *MoC*. L'utilisateur final (expert du domaine) n'a plus à étudier, apprendre et maîtriser un *MoC* pour exprimer les aspects concurrents d'un système, puisque ceux-ci sont déjà capturés au niveau des constructions du langage. Cette tâche revient donc au concepteur du langage qui doit exprimer, dans la sémantique d'exécution, les aspects concurrents des éléments de la syntaxe abstraite du langage.

Nous avons illustré dans le Chapitre 3 le principe de la séparation des préoccupations entre les aspects séquentiels (règles sémantiques – *Semantic Rules*) et les aspects concurrents (*Model of Concurrency Mapping – MoCMapping*). Puis nous avons étudié comment rendre possible ou simplifier l'expression de certaines constructions de langages. Cela nous a conduit en particulier à raffiner les règles sémantiques et le protocole de communication (*Communication Protocol*). Dans le Chapitre 4, nous avons expliqué comment le *MoC* originellement utilisé dans l'approche, *Event Structures*, n'est pas le plus adapté pour tous les xDSMLs. Pour palier cela, nous avons présenté une vision récursive de notre approche, permettant l'utilisation en tant que *MoC* de n'importe quel *concurrency-aware xDSML* ayant été défini précédemment. Enfin, dans le Chapitre 5 nous avons présenté comment spécifier la sémantique d'exécution de façon translationnelle. En particulier, nous avons expliqué comment rendre possible l'exécution de tels langages de la même façon que les langages dont la sémantique est définie de manière opérationnelle.

Il reste malgré tout de nombreux aspects qui peuvent être le sujet de travaux de recherche ultérieurs. Par exemple, dans la continuité de ce que nous avons présenté au Chapitre 4, nous souhaiterions fournir une bibliothèque standard de MoCs, comprenant par exemple les réseaux de Pétri [107, 71] et le modèle d'acteur [65]. Nous aimerions aussi avoir la

possibilité de vérifier des propriétés comportementales pour les systèmes représentés avec un *concurrency-aware xDSML* dont la sémantique est spécifiée de façon translationnelle. Ceci implique d'arriver à traduire les propriétés dans un premier temps, puis d'arriver à traduire les résultats de l'analyse dans un second temps [163, 162]. L'approche que nous avons décrite se concentre sur les aspects *spécification* des langages, et non sur leur *implémentation*. Elle n'est donc pas adaptée pour les *xDSMLs* ayant d'importants besoins de performance d'exécution. À la place, une génération de code optimisé pourrait être mise en place. Nous pourrions aussi améliorer la gestion des points de variation sémantique (*Semantic Variation Points – SVPs*) en utilisant des techniques de gestion de la variabilité déjà connues [148]. Enfin, il faudrait aussi pouvoir facilement intégrer les *xDSMLs* entre eux via l'extension de la syntaxe abstraite (possible par l'héritage par exemple) et surtout via l'extension des aspects sémantiques (redéfinition des *Execution Functions*, héritage de *MoCMapping*, héritage de *Communication Protocol*, etc.). Ceci permettrait de plus facilement partager des bouts de langages communs à de nombreux *xDSMLs*, comme les expressions arithmétiques.

6.1 Conclusion

Language design is placed at the heart of the software engineering process by Language-Oriented Programming (LOP), a new paradigm brought to life with the goal of keeping up with the complexity of modern highly concurrent softwares and systems and the increasingly-parallel platforms on which they are executed. Meanwhile, Models of Concurrency (MoCs) have been developed to formalize the concurrent aspects of these systems, enabling their late specialization to a specific execution platform, therefore allowing the domain expert to focus on their area of expertize in the system design activities.

This thesis has focused on bridging the gap between Language-Oriented Programming (LOP) and Models of Concurrency (MoCs) through the specification of languages which systematically use a MoC to describe the concurrent aspects of a system. Such languages are called concurrency-aware eXecutable Domain-Specific Modeling Languages (xDSMLs). The systematic use of a MoC offers the following upsides. First, an appropriate formalism is used to express the concurrent features of the language, thus facilitating its specification, implementation and debugging. The domain-specificity of the language which allows xDSMLs to *systematically* use a certain MoC also contributes to helping the end user of the language, since they do not need to learn about the specific MoC, or its implementation and associated good practices, to reap its benefits. The MoC is automatically used for any system defined using the concurrency-aware xDSML, and the resulting model-level specification (instance of the MoC used) represents the concurrency concerns of the model. It can be analyzed to ensure behavioral properties of the system being developed, depending on the MoC used. The xDSML can also be refined for a specific execution platform without affecting the other concerns of the semantics. The use of a MoC is thus not an addition to the execution semantics of the language, but rather a re-organization of the semantics through a clear separation of concerns. This modularity helps when debugging a language, or when considering its semantic variants.

These benefits come at the cost of a complicated language design activity. The operational execution semantics of concurrency-aware xDSMLs are separated into its concurrent aspects (Model of Concurrency Mapping – MoCMapping) and its data aspects (Semantic Rules), coordinated by a Communication Protocol. Identifying what parts of the semantics belong to which concern can be difficult, so is identifying the right metalanguage for each concern, and so is their use by the language designer. Still, this modularity benefits the language designer on the long term, since each concern can be developed, refined and tested independently. They can also be reused to create semantic variations of an xDSML.

Since these concerns would have been disseminated throughout the whole semantics, this identification would have taken place anyway when trying to refine a language.

In Chapter 3, we have detailed the initial approach, which consists in separating the concerns in the operational execution semantics of xDSMLs. We then identified its shortcomings, usually related to the expressive power or complexity of specifying some xDSMLs, or related to the difficulty of how they can be specified. Thus, we have proposed several features pertaining to the design of the Semantic Rules, the coordination of the Semantic Rules and the MoCMapping, and the runtime of concurrency-aware xDSMLs.

In Chapter 4, we have enriched the concurrency-aware xDSML approach by enabling new MoCs to be defined and integrated. More precisely, we have given a recursive definition of the approach, thus enabling concurrency-aware xDSMLs to be used as MoCs. This contribution gives a common interface to MoCs (*i.e.*, as concurrency-aware xDSMLs), while simplifying the MoCMapping (based on a transformation instead of on a MoC-specific metalanguage). It favors the use of an adequate MoC for each xDSML. In this proposal, only the concurrency concerns of the xDSML being defined are translated to another concurrency-aware xDSML.

In Chapter 5, we have proposed an alternative means to specify the execution semantics of concurrency-aware xDSMLs, by using a translational approach instead of an operational one. We have described how additional specifications are required in order for the execution of a translationally-defined concurrency-aware xDSML to be up to par with the execution of an operationally-defined one. Translational semantics allow the full reuse of a previously-defined xDSML, whose semantics and associated tools are already available.

Our contributions have been motivated and illustrated on example xDSMLs and models, and most of them (*i.e.*, most of Chapter 3, Chapter 4) have been implemented in the GEMOC Studio, an Eclipse-based language workbench on top of implementation of modeling standards from the OMG.

Overall, in our thesis, we have proposed and experimented several approaches participating in the implementation of the semantics of concurrency-aware xDSMLs, which bring together Language-Oriented Programming and Models of Concurrency in a synergetic language design approach. This approach exposes an explicit behavioral interface for the xDSMLs, through the Mappings defined in the Communication Protocol. We have also proposed in Chapter 3 a means to define Composite Mappings, which contribute to a higher-level view of the behavioral semantics. These can be exploited to orchestrate, at the language level, the coordinated execution of models conforming to different concurrency-aware xDSMLs. This was concretized during the ANR INS GEMOC Project in a metalanguage, the Behavioral Coordination Operator Language (B-COol) [154]. It reifies co-

ordination patterns between xDSMLs. Its development and use are out of scope of our work, but were the subject of another PhD thesis [153] during the time of the project. This approach can contribute to tackling the complexity of designing and developing the highly-concurrent softwares and systems of tomorrow such as Cyber-Physical Systems, the Internet of Things, or Smart Cities.

6.2 Perspectives

We have identified several possible future research directions to improve the concurrency-aware xDSML approach.

First, we would like to ease the use of the concurrency-aware approach by providing a standard library of MoCs, based on the recursive definition we have given of the approach in Chapter 4. MoCs are usually defined informally (*i.e.*, using natural language), and each implementation brings its own flavors of details. In order to facilitate the specification of xDSMLs using MoCs other than Event Structures, while still using formalisms well-known by computer scientists, we believe the approach should provide default implementations for classical MoCs such as Petri nets or the Actor model. By providing such MoCs, we could also integrate the use of associated verification tools (*i.e.*, model checkers, etc.) for well-known MoCs.

When considering concurrency-aware xDSMLs with translational semantics, any analysis performed on the concurrent concerns will be pertinent to the target model, and not to the source model. Our approach could be improved by providing the means to specify properties for the source model, verified on the target model, and with meaningful results being expressed for the source model [162]. In the same spirit, this could also be done in the context of the recursive definition of concurrency-aware xDSMLs (*cf.* Chapter 4), except that the "target model" only captures the concurrency concerns of the source model.

By making explicit the concurrency concerns of a language, we have added an extra stage in its specification, and also in its runtime. This may be problematic for xDSMLs with a focus on the performance of their runtime. In particular, xDMSLs defined using our recursive approach would require the coordination of various runtimes, possibly up to a point where the execution of a model would become unpractical or too expensive. This is not a problem in the context in which our work was done, since we targeted the specification, and not the implementation, of xDSMLs. Still, further study of the runtime costs associated with the concurrency-aware approach, both for larger models and for larger languages, could be performed to identify the physical limitations of the approach.

One way to solve this issue would be to generate optimized code based on the artefacts used for the execution of a model.

Some languages put an emphasis on the notion of *time*. In the concurrency-aware approach, one must make the distinction between what we call the *logical time* (in the model-level instance of the MoC), the *domain time* (captured in the abstract syntax), and the *physical time* (in the runtime of the Semantic Rules). Identifying which notion is relevant to an xDSML, and how it can be coordinated with the relevant parts of the specification would require further work.

We have mentioned how Semantic Variation Points (SVPs) can be specified and implemented in operational semantics, in particular when they pertain to the concurrency concerns of the language. They manifest themselves as points of nondeterminism which, unless specialized, are resolved heuristically by the runtime. Implementing SVPs typically consists in resolving (possibly, only partially) (some of) these nondeterminisms. However, for some languages, non-deterministicity is a feature of the language semantics which should not be overridden by dialects. Therefore, the MoCMapping should include the possibility to hinder some of its parts from being extended by dialects. This may be difficult to specify when relying on the recursive definition we have proposed in Chapter 4, since the MoCMapping is implemented by an Abstract Syntax Transformation. Additional work should study this possibility.

We have described how SVPs can be implemented, but they must also be managed. For instance, being able to cherry-pick specific implementations of individual SVPs could ease the management of the variability of an xDSML. Many variability management techniques could be applied to SVPs in the context of concurrency-aware xDSMLs [148]. Additionally, the sharing of language parts could be streamlined. For instance, many xDSMLs may need to rely on arithmetic expressions, in which case it would be better to only maintain one expression language which could easily be integrated into a wider-scope xDSML. To do so, all the language parts should provide some mechanisms of extension. This is the case of inheritance for the Abstract Syntax, but may be more tricky considering the different parts related to the execution semantics. Since the Semantic Rules are weaved onto the Abstract Syntax, traditional extension mechanisms could be applied like the redefinition of operations, the reuse of the super implementation, etc. For the MoCMapping, when using Event Structures, the symbolic partial orderings are extensible as mentioned in Section 3.8: a partial ordering can easily be inserted into another partial ordering. However, the notion of extension between transformations is harder to identify. As for the Communication Protocol, an extension mechanism similar to inheritance remains to be defined.

A metalanguage such as Melange [30], which focuses on composing DSL parts, could be used for this purpose.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] Apache Software Foundation. Apache Flink, 2016. URL https://flink.apache.org/.

[3] Apache Software Foundation. Apache Storm, 2016. URL https://storm.apache.org/.

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[5] Daniel Balasubramanian, Corina S Păsăreanu, Michael W Whalen, Gábor Karsai, and Michael Lowry. Polyglot: Modeling and Analysis for Multiple Statechart Formalisms. In *ISSTA*. ACM, 2011.

[6] Hendrik Pieter Barendregt. *The Lambda Calculus*, volume 3. North-Holland Amsterdam, 1984.

[7] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[8] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. *SIGPLAN Not.*, 24(2):14–24, November 1988. ISSN 0362-1340. doi: 10.1145/64140.65005. URL http://doi.acm.org/10.1145/64140.65005.

[9] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Elyes Rouis. Modeling heterogeneous points of view with modhel'x. In Sudipto Ghosh, editor, *Models in Software Engineering: Workshops and Symposia at MoDELS 2009, Denver, CO, USA, October 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2010. ISBN 978-3-642-12260-6. doi: http://dx.doi.org/10.1007/978-3-642-12261-3_29.

[10] Phillip J Brooke, Richard F Paige, and Jeremy L Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.

[11] Paul Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. The Pragmatic Programmers, 2014.

[12] Fabien Campagne. *The MPS Language Workbench*, volume 1. Fabien Campagne, 2014.

[13] Minder Chen and Jay F Nunamaker. MetaPlex: An Integrated Environment for Organization and Information System Development. In *International Conference on Information Systems: Proceedings of the tenth international conference on Information Systems: Boston, Massachusetts, United States*, volume 1989, pages 141–151, 1989.

[14] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of mathematics*, pages 346–366, 1932.

[15] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle – Application à l'ingénierie des procédés*. PhD thesis, Institut National Polytechnique de Toulouse, Université de Toulouse, July 2008. URL http://ethesis.inp-toulouse.fr/archive/00000666/. In French.

[16] Benoit Combemale. Towards Language-Oriented Modeling, 2015. URL https://hal.inria.fr/tel-01238817. Habilitation à Diriger des Recherches (HDR).

[17] Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *19th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 282–287. IEEE, 2012.

[18] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *5th International Conference on Software Language Engineering (SLE2012)*, LNCS. Springer, September 2012. URL http://hal.inria.fr/hal-00725643.

[19] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, 'Etats-Unis, 2013. Springer-Verlag. URL http://hal.inria.fr/hal-00850770.

[20] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B France, Jean-Marc Jézéquel, and Jordan Gray. Globalizing Modeling Languages. *Computer*, 47(6):68–71, 2014.

[21] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Pearson Education, 2007.

[22] Michelle L Crane and Jürgen Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. *Software & Systems Modeling*, 6(4):415–435, 2007.

[23] Gianpaolo Cugola and Alessandro Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[24] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. USA, 2003.

[25] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

[26] Juan De Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Metamodelling. In *FASE*, volume 2, pages 174–188. Springer, 2002.

[27] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Technical report, Inria, 2012.

[28] Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *Objects, Models, Components, Patterns*, pages 34–41. Springer, 2012.

[29] Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoit Combemale. Towards a Meta-Language for the Concurrency Concern in DSLs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, Grenoble, France, March 2015. URL https://hal.inria.fr/hal-01087442.

[30] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 25–36, 2015. doi: 10.1145/2814251.2814252. URL http://doi.acm.org/10.1145/2814251.2814252.

[31] Edsger W Dijkstra. A Simple Axiomatic Basis for Programming Language Constructs. *Indagationes Mathematicae (Proceedings)*, 77(1):1–15, 1974. ISSN 1385-7258. doi: http://dx.doi.org/10.1016/1385-7258(74)90008-0. URL http://www.sciencedirect.com/science/article/pii/1385725874900080.

[32] DIVERSE-team. GitHub for Kermeta 3, 2016. URL http://github.com/diverse-project/k3/.

[33] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm, 2004. URL http://www.onboard.jetbrains.com/articles/04/10/lop/mps.pdf.

[34] Eclipse Foundation. Ecore on the Eclipse wiki, 2010. URL http://wiki.eclipse.org/Ecore.

[35] Eclipse Foundation. Object Constraint Language (OCL) on the Eclipse Wiki, 2016. URL http://wiki.eclipse.org/OCL.

[36] Eclipse Foundation. Eclipse Modeling Framework (EMF) Homepage, 2016. URL http://www.eclipse.org/modeling/emf/.

[37] Eclipse Foundation. Sirius Homepage, 2016. URL https://www.eclipse.org/sirius/.

[38] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 112–121, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371419. URL http://doi.acm.org/10.1145/2371401.2371419.

[39] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The State of the Art in Language Workbenches. In *Software language engineering*, pages 197–217. Springer, 2013.

[40] EsperTech. Esper, 2016. URL http://www.espertech.com/esper/.

[41] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[42] Matthias Felleisen. On the Expressive Power of Programming Languages. In *ESOP'90*, pages 134–151. Springer, 1990.

[43] David Fetter. High Performance SQL with PostgreSQL 8.4, 2009. URL http://assets.en.oreilly.com/1/event/27/High%20Performance%20SQL%20with%20PostgreSQL%20Presentation.pdf.

[44] Cédric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385. ACM, 1996.

[45] Martin Fowler. Language Workbenches: The Killer-app for Domain Specific Languages, 2005. URL http://martinfowler.com/articles/languageWorkbench.html.

[46] Martin Fowler. Projectional Editing, 2008. URL http://martinfowler.com/bliki/ProjectionalEditing.html.

[47] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[48] Romain Franceschini, Paul-Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia, and David Hill. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *OASIcs-OpenAccess Series in Informatics*, volume 43. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[49] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., 2010.

[50] Holger Giese, Tihamér Levendovszky, and Hans Vangheluwe. Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. In *Models in Software Engineering*, pages 252–262. Springer, 2006.

[51] Brian Goetz and Tim Peierls. *Java Concurrency in Practice*. Pearson Education, 2006.

[52] Google. Golang, 2016. URL http://golang.org/.

[53] Richard C Gronback. *Eclipse Modeling Project: a Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009.

[54] The Liberty Research Group. Glossary of Modeling Terms, 2016. URL http://liberty.princeton.edu/Research/Modeling/glossary.php.

[55] Munish Gupta. *Akka Essentials*. Packt Publishing Ltd, 2012.

[56] Vineet Gupta. *CHU Spaces: a Model of Concurrency*. PhD thesis, stanford university, 1994.

[57] Nicolas Halbwachs. A Synchronous Language at Work: the Story of Lustre. In *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.*, pages 3–11, 2005.

[58] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[59] Cécile Hardebolle and Frédéric Boulanger. Exploring Multi-Paradigm Modeling Techniques. *Simulation*, 85(11-12):688–708, 2009.

[60] Cécile Hardebolle and Frédéric Boulanger. Simulation of Multi-Formalism Models with ModHel'X. In *Proceedings of the first IEEE International Conference on Software Testing Verification and Validation*, 2008.

[61] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of computer programming*, 8(3):231–274, 1987.

[62] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 2004.

[63] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *TOSEM*, 5(4):293–333, 1996.

[64] Carl Hewitt and H Zenil. What is Computation? Actor Model versus Turing's Model. *A Computable Universe: Understanding and Exploring Nature as Computation*, pages 159–85, 2013.

[65] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[66] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 480–483. IEEE, 2011.

[67] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[68] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Springer, 1978.

[69] Papa Issa Diallo, Joël Champeau, and Vincent Leilde. Model Based Engineering for the support of Models of Computation: The Cometa Approach. In *International Workshop on Multi-Paradigm Modeling - MPM 2011*, volume 42, pages ISSN 1863–2122, Wellington, New Zealand, October 2011. URL https://hal-ensta-bretagne.archives-ouvertes.fr/hal-00635594.

[70] JBoss Community Documentation. Drools Complex Event Processing, 2016. URL https://docs.jboss.org/drools/release/6.1.0.Final/drools-docs/html/DroolsComplexEventProcessingChapter.html.

[71] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer Science & Business Media, 1997.

[72] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer, 2009.

[73] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.

[74] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of computer programming*, 72(1):31–39, 2008.

[75] ISO/IEC JTC1/SC22/WG14. C Programming Language Standard (C11) ISO/IEC 9899. Standard, International Organization for Standardization (ISO), International Electrotechnical Comission (IEC), 2011.

[76] Gilles Kahn. *Natural Semantics*. Springer, 1987.

[77] Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. URL http://doi.acm.org/10.1145/1869459.1869497.

[78] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[79] Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.

[80] Tomaž Kosar, Pablo E Martı, Pablo A Barrientos, Marjan Mernik, et al. A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. *Information and Software Technology*, 50(5):390–405, 2008.

[81] Tomaž Kosar, Marjan Mernik, Mastej Črepinšek, Pedro Rangel Henriques, Daniela Da Cruz, Maria João Varanda Pereira, and Nuno Oliveira. Influence of Domain-Specific Notation to Program Understanding. In *Computer Science and Information Technology, 2009. IMCSIT'09. International Multiconference on*, pages 675–682. IEEE, 2009.

[82] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[83] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Oyvind Bech, Anders Sandven, and Adrian Rutle. DPF Workbench: a multi-level language workbench for MDE. *Proceedings of the Estonian Academy of Sciences*, 62(1):3–15, 2013.

[84] Language Workbenches Challenge Committee. Language Workbenches Challenge: Comparing Tools of the Trade, 2011. URL http://www.languageworkbenches.net/.

[85] Florent Latombe, Xavier Crégut, Benoît Combemale, Julien Deantoni, and Marc Pantel. Weaving Concurrency in eXecutable Domain-Specific Modeling Languages. In *8th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*, Pittsburg, United States, October 2015. ACM. URL https://hal.inria.fr/hal-01185911.

[86] Florent Latombe, Xavier Crégut, Julien Deantoni, Marc Pantel, and Benoit Combemale. Coping with Semantic Variation Points in Domain-Specific Modeling Languages. In *1st International Workshop on Executable Modeling (EXE 2015)*, Ottawa, Canada, 2015. CEUR. URL https://hal.inria.fr/hal-01222999.

[87] Florent Latombe, Xavier Crégut, and Marc Pantel. Concurrency-aware eXecutable Domain-Specific Modeling Languages as Models of Concurrency. In *2nd International Workshop on Executable Modeling (EXE 2016)*, Saint-Malo, France, 2016. CEUR. URL https://hal.inria.fr/hal-01357001.

[88] Edward Lee, Alberto Sangiovanni-Vincentelli, et al. A Framework for Comparing Models of Computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[89] Edward A Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[90] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, Java SE 8 Edition, 2015. URL https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[91] David Luckham. *The Power of Events*, volume 204. Addison-Wesley Reading, 2002.

[92] Frédéric Mallet. Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 2008. doi: 10.1007/s11334-008-0055-2.

[93] MathWorks. Simulink, 2016. URL http://www.simulink.com/.

[94] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *Software Language Engineering*, pages 56–75. Springer, 2013.

[95] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[96] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and How to Develop Domain-Specific Languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[97] Bertrand Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

[98] Bertrand Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, 1993.

[99] Microsoft. Microsoft StreamInsight, 2016. URL https://msdn.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx.

[100] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Halsted Press, 1977.

[101] Robin Milner. *A Calculus of Communicating Systems*. Berlin; New York: Springer-Verlag, 1980.

[102] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge university press, 1999.

[103] Peter D Mosses. Modular Structural Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60:195–228, 2004.

[104] Pieter J Mosterman and Hans Vangheluwe. Guest Editorial: Special Issue on Computer Automated Multi-Paradigm Modeling. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):249–255, 2002.

[105] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Model Driven Engineering Languages and Systems*, pages 264–278. Springer, 2005.

[106] Mogens Nielsen. Models for Concurrency. In *Mathematical Foundations of Computer Science 1991*, pages 43–46. Springer, 1991.

[107] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science*, 13(1):85 – 108, 1981. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(81)90112-2. URL http://www.sciencedirect.com/science/article/pii/0304397581901122. Special Issue Semantics of Concurrent Computation.

[108] Mogens Nielsen, Vladimiro Sassone, and Glynn Winskel. *Relationships between Models of Concurrency*. Springer, 1994.

[109] Object Management Group (OMG). MOF Model To Text Transformation Language Specification v1.0, 2008. URL http://www.omg.org/spec/MOFM2T/.

[110] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) Specification v1.1, 2011. URL http://www.omg.org/spec/MARTE/.

[111] Object Management Group (OMG). Unified Modeling Language (UML) Superstructure Specification v2.4.1, 2011. URL http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[112] Object Management Group (OMG). Meta-Object Facility (MOF) Specification v2.5, 2015. URL http://www.omg.org/spec/MOF/.

[113] Object Management Group (OMG). Object Constraint Language (OCL) Specification v2.4, 2015. URL http://www.omg.org/spec/OCL/.

[114] Object Management Group (OMG). Query/View/Transformation (QVT) Specification v1.2, 2015. URL http://www.omg.org/spec/QVT/.

[115] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification v2.5.1, 2015. URL http://www.omg.org/spec/XMI/.

[116] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (fUML) Specification v1.2.1, 2016. URL http://www.omg.org/spec/FUML/.

[117] Object Management Group (OMG). Model-Driven Architecture (MDA), 2016. URL http://www.omg.org/mda/.

[118] OpenJDK. HotSpot Runtime Overview, 2015. URL http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline.

[119] Oracle. Oracle Complex Event Processing, 2016. URL https://docs.oracle.com/cd/E13157_01/wlevs/docs30/get_started/overview.html.

[120] Chun Ouyang, Eric Verbeek, Wil MP Van Der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM Ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(2):162–198, 2007.

[121] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[122] Adrian Paschke, Paul Vincent, and Florian Springer. Standards for Complex Event Processing and Reaction Rules. In *Rule-Based Modeling and Computing on the Semantic Web*, pages 128–139. Springer, 2011.

[123] Maria João Varanda Pereira, Marjan Mernik, Pedro Rangel Henriques, et al. Program Comprehension for Domain-Specific Languages. *Computer Science and Information Systems*, 5(2):1–17, 2008.

[124] Robert G Pettit IV and Navneet Mezcciani. Highlighting the Challenges of Model-Based Engineering for Spaceflight Software Systems. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pages 51–54. IEEE Press, 2013.

[125] Gordon D Plotkin. The Origins of Structural Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[126] Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[127] Claudius Ptolemaeus. *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org Berkeley, CA, USA, 2014.

[128] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-based Editors*. Springer Science & Business Media, 2012.

[129] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[130] Jon G Riecke. Fully Abstract Translations Between Functional Languages. *Mathematical Structures in Computer Science*, 3(04):387–415, 1993.

[131] Jose E Rivera and Antonio Vallecillo. Adding Behavior to Models. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 169–169. IEEE, 2007.

[132] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[133] Nick Russell, Arthur H. M. Ter Hofstede, and Nataliya Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, Queensland University of Technology and Eindhoven University of Technology, 2006.

[134] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

[135] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit—a Flexible Graphical Environment for Methodology Modelling. In *Advanced Information Systems Engineering*, pages 168–193. Springer, 1991.

[136] Eiffel Software. Eiffel Software Homepage, 2016. URL https://www.eiffel.com/.

[137] Paul G Sorenson, Jean-Paul Tremblay, and Andrew J McAllister. The Metaview System for Many Specification Environments. *IEEE software*, 5(2):30, 1988.

[138] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and transformational techniques in software engineering II*, pages 408–424. Springer, 2008.

[139] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software & Systems Modeling*, 9(1):7–20, 2010.

[140] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. ÆMINIUM: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.*, 36(1): 2:1–2:42, March 2014. ISSN 0164-0925. doi: 10.1145/2543920. URL http://doi.acm.org/10.1145/2543920.

[141] Martin Sústrik. Structured Concurrency, 2016. URL http://250bpm.com/blog:71.

[142] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[143] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*, pages 21–36. Springer, 2012.

[144] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why Do Scala Developers Mix the Actor Model with other Concurrency Models? In *ECOOP 2013–Object-Oriented Programming*, pages 302–326. Springer, 2013.

[145] Toby J Teorey. *Database Modeling & Design*. Morgan Kaufmann, 1999.

[146] Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[147] Parallel Universe. Quasar, 2016. URL http://docs.paralleluniverse.co/quasar/.

[148] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoit Combemale. Variability Support in Domain-Specific Language Development. In *Software Language Engineering*, pages 76–95. Springer, 2013.

[149] Antonio Vallecillo. On the Combination of Domain Specific Modeling Languages. In *Modelling Foundations and Applications*, pages 305–320. Springer Berlin Heidelberg, 2010.

[150] Mark GJ van den Brand, Arie van Deursen, Jan Heering, HA De Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A Olivier, Jeroen Scheerder, et al. The ASF+ SDF Meta-Environment: A Component-Based Language Development Environment. In *Compiler Construction*, pages 365–370. Springer, 2001.

[151] Tijs van der Storm. *The Rascal Language Workbench*. CWI. Software Engineering [SEN], 2011.

[152] Peter Van Roy et al. Programming Paradigms for Dummies: What Every Programmer Should Know. *New computational paradigms for computer music*, 104, 2009.

[153] Matias Ezequiel Vara Larsen. *BCool: the Behavioral Coordination Operator Language*. Phd thesis, Université de Nice Sophia Antipolis, April 2016. URL https://hal.inria.fr/tel-01302875.

[154] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Ottawa, Canada, September 2015. ACM. URL https://hal.inria.fr/hal-01182773. to be published in the proceedings of the Models 2015 conference.

[155] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook. org, 2013.

[156] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[157] Martin P Ward. Language-Oriented Programming. *Software-Concepts and Tools*, 15 (4):147–161, 1994.

[158] Stephen A White. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies Inc., 2008.

[159] Jeannette M Wing. FAQ on π-Calculus. *Microsoft Internal Memo*, 2002.

[160] Glynn Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS. Springer, 1987.

[161] WSO2. WSO2 Complex Event Processor, 2016. URL http://wso2.com/products/complex-event-processor/.

[162] Faiez Zalila. *Methods and Tools for the Integration of Formal Verification in Domain-Specific Languages*. PhD thesis, Institut National Polytechnique de Toulouse, Université de Toulouse, December 2014. URL http://oatao.univ-toulouse.fr/14159/.

[163] Faiez Zalila, Xavier Crégut, and Marc Pantel. A Transformation-Driven Approach to Automate Feedback Verification Results. In *Model and Data Engineering*, pages 266–277. Springer, 2013.

Appendices

Appendices Outline

Summary of Appendices

- Appendix A lists all the possible execution scenarios of the example fUML Activity presented in Chapter 3. It takes into account the different possible schedulings of the concurrent branches of the `ForkNode`, the different possibilities following the `DecisionNode`, and the different order of evaluations for the guards of its outgoing edges.
- Appendix B shows the different specifications composing the concurrency-aware definition of fUML in the GEMOC Studio.
- Appendix C details the execution and graphical animation of the example fUML Activity in the GEMOC Studio.
- Appendix D gives the different specifications composing the concurrency-aware definition of the Threading xDSML presented in Chapter 4, using the GEMOC Studio.
- Appendix E details the new specifications used for the concurrency-aware definition of fUML in the GEMOC Studio, using the Threading xDSML as MoC (whose implementation is shown in Appendix D) as presented in Chapter 4.
- Appendix F shows the execution of the example fUML Activity, in the GEMOC Studio, using the fUML specification based on the Threading xDSML as its MoC (*cf.* Appendix E), as described in Chapter 4.
- Appendix G gives the Xtext textual concrete syntax of the Communication Protocol metalanguage, the GEMOC Events Language (GEL), described in Section 3.11.
- Appendix H gives the Xtext textual concrete syntax of the Projections metalanguage described in Chapter 4.

Résumé des Annexes

- Annexe A énumère l'ensemble des scénarios d'exécution possible pour l'activité fUML utilisée en exemple et présentée initialement dans le Chapitre 3. Leur nombre important est dû à la concurrence entre les branches, et au fait que les gardes à la sortie d'un noeud *DecisionNode* peuvent être évaluées dans n'importe quel ordre.
- Annexe B contient la spécification du langage fUML, selon l'approche *concurrency-aware xDSML* présentée dans le Chapitre 3, à l'aide du GEMOC Studio.
- Annexe C détaille l'exécution pas-à-pas et l'animation graphique, dans le GEMOC Studio, du modèle d'activité fUML utilisé comme exemple.
- Annexe D contient la spécification du langage *Threading* présenté dans le Chapitre 4. Il implémente le modèle de *threads* traditionnellement utilisé par les langages de programmation tels Java ou Python. Cette spécification est réalisée à l'aide du GEMOC Studio.
- Annexe E détaille la spécification de fUML à l'aide du *xDSML* de *Threading* (dont l'implémentation est présentée dans l'Annexe D) selon l'approche décrite dans le Chapitre 4.
- Annexe F illustre l'exécution, dans le GEMOC Studio, de l'exemple d'activité fUML à l'aide de la définition de fUML présentée dans l'Annexe E.
- Annexe G contient la définition de la syntaxe concrète textuelle Xtext de GEL, le métalangage utilisé pour la spécification du protocol de communication (*Communication Protocol*) d'un *concurrency-aware xDSML*. Ce méta-langage est présenté et décrit dans la section 3.11.
- Annexe H présente la définition de la syntaxe concrète textuelle Xtext de notre implémentation du métalangage pour définir les *Projections* entre un xDSML et son MoC, comme décrit dans le Chapitre 4.

A

Enumeration of the Possible Execution Scenarios of the Example fUML Activity

Figure A.1 shows the example Activity used to illustrate fUML in Chapters 3, 4 and 5. To consider all the possible execution scenarios of this Activity, we must consider the following details of the fUML Semantics [116, 111]:

- Concurrent branches (*i.e.*, between a ForkNode and its corresponding JoinNode) can be executed in sequence, in parallel or using any possible interleavings between their respective contents

- For a DecisionNode, the guards can be evaluated in any order, possibly even in parallel.

An important point to take into consideration is that we will enumerate the different possible scenarios, *independently from the runtime data of the model* (*i.e.*, from the result of the evaluation of each guard). Otherwise, a lot of scenarios are duplicated due to the DecisionNode which splits each existing scenario into three depending on which branch is executed. Our focus is on how concurrent elements of an fUML Activity are scheduled, not in the semantics of a Decision Node. Therefore, to enumerate all possible scenarios

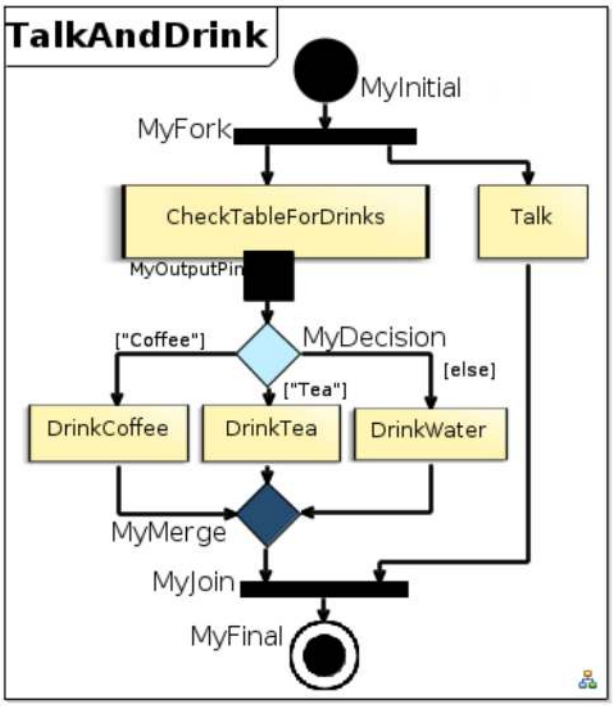

Figure A.1: Example fUML activity where we want to drink something from the table while talking.

while taking the data of the model into account, one would have to multiply the number of scenarios before anything is drunk by the number of possible drinks (3).

Below, we will use the following syntax, considering two nodes A and B:

- $A \rightarrow B$ designates the *sequence* of executing A and then executing B
- $A \mid B$ designates the *parallel* execution of A and B

We first consider the evaluation of the guards and the drinking of a drink as regular nodes "[Evaluation]" and "[Drinking]" which we will detail later on.

First, "Talking" may happen in parallel with any of the nodes of the drinking part of the activity, therefore we have the 6 following scenarios:

1. CheckTableForDrinks | **Talking** → OutputPin → DecisionNode → [Evaluation] → [Drinking] → MergeNode
2. CheckTableForDrinks → OutputPin | **Talking** → DecisionNode → [Evaluation] → [Drinking] → MergeNode

3. CheckTableForDrinks → OutputPin → DecisionNode | **Talking** → [Evaluation] → [Drinking] → MergeNode
4. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] | **Talking** → [Drinking] → MergeNode
5. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → [Drinking] | **Talking** → MergeNode
6. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → [Drinking] → MergeNode| **Talking**

It may also happen interleaved with any of the nodes of the drinking part, thus we have 7 additional scenarios:

1. **Talking** → CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → [Drinking] → MergeNode
2. CheckTableForDrinks → **Talking** → OutputPin → DecisionNode → [Evaluation] → [Drinking] → MergeNode
3. CheckTableForDrinks → OutputPin → **Talking** → DecisionNode → [Evaluation] → [Drinking] → MergeNode
4. CheckTableForDrinks → OutputPin → DecisionNode → **Talking** → [Evaluation] → [Drinking] → MergeNode
5. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → **Talking** → [Drinking] → MergeNode
6. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → [Drinking] → **Talking** → MergeNode
7. CheckTableForDrinks → OutputPin → DecisionNode → [Evaluation] → [Drinking] → MergeNode → **Talking**

Now let us detail the "[Evaluation]" bit. There are three guards, which can be evaluated in any order, possibly in parallel. Thus, "[Evaluation] | **Talking**" can be detailed as being one of the followings:

1. GuardForCoffee | GuardForTea | GuardForWater | **Talking**
2. GuardForCoffee | GuardForTea | **Talking** → GuardForWater

3. GuardForCoffee | GuardForTea → **Talking** → GuardForWater
4. GuardForCoffee | GuardForTea → GuardForWater | **Talking**
5. GuardForCoffee | GuardForWater | **Talking** → GuardForTea
6. GuardForCoffee | GuardForWater → **Talking** → GuardForTea
7. GuardForCoffee | GuardForWater → GuardForTea | **Talking**
8. GuardForTea | GuardForWater | **Talking** → GuardForCoffee
9. GuardForTea | GuardForWater → **Talking** → GuardForCoffee
10. GuardForTea | GuardForWater → GuardForCoffee | **Talking**
11. GuardForCoffee | **Talking** → GuardForTea | GuardForWater
12. GuardForCoffee → **Talking** → GuardForTea | GuardForWater
13. GuardForCoffee → GuardForTea | GuardForWater | **Talking**
14. GuardForTea | **Talking** → GuardForCoffee | GuardForWater
15. GuardForTea → **Talking** → GuardForCoffee | GuardForWater
16. GuardForTea → GuardForCoffee | GuardForWater | **Talking**
17. GuardForWater | **Talking** → GuardForCoffee | GuardForTea
18. GuardForWater → **Talking** → GuardForCoffee | GuardForTea
19. GuardForWater → GuardForCoffee | GuardForTea | **Talking**
20. GuardForCoffee | **Talking** → GuardForTea → GuardForWater
21. GuardForCoffee → **Talking** → GuardForTea → GuardForWater
22. GuardForCoffee → GuardForTea | **Talking** → GuardForWater
23. GuardForCoffee → GuardForTea → **Talking** → GuardForWater
24. GuardForCoffee → GuardForTea → GuardForWater | **Talking**
25. GuardForCoffee | **Talking** → GuardForWater → GuardForTea
26. GuardForCoffee → **Talking** → GuardForWater → GuardForTea

27. GuardForCoffee → GuardForWater | **Talking** → GuardForTea
28. GuardForCoffee → GuardForWater → **Talking** → GuardForTea
29. GuardForCoffee → GuardForWater → GuardForTea | **Talking**
30. GuardForTea | **Talking** → GuardForCoffee → GuardForWater
31. GuardForTea → **Talking** → GuardForCoffee → GuardForWater
32. GuardForTea → GuardForCoffee | **Talking** → GuardForWater
33. GuardForTea → GuardForCoffee → **Talking** → GuardForWater
34. GuardForTea → GuardForCoffee → GuardForWater | **Talking**
35. GuardForTea | **Talking** → GuardForWater → GuardForCoffee
36. GuardForTea → **Talking** → GuardForWater → GuardForCoffee
37. GuardForTea → GuardForWater | **Talking** → GuardForCoffee
38. GuardForTea → GuardForWater → **Talking** → GuardForCoffee
39. GuardForTea → GuardForWater → GuardForCoffee | **Talking**
40. GuardForWater | **Talking** → GuardForCoffee → GuardForTea
41. GuardForWater → **Talking** → GuardForCoffee → GuardForTea
42. GuardForWater → GuardForCoffee | **Talking** → GuardForTea
43. GuardForWater → GuardForCoffee → **Talking** → GuardForTea
44. GuardForWater → GuardForCoffee → GuardForTea | **Talking**
45. GuardForWater | **Talking** → GuardForTea → GuardForCoffee
46. GuardForWater → **Talking** → GuardForTea → GuardForCoffee
47. GuardForWater → GuardForTea | **Talking** → GuardForCoffee
48. GuardForWater → GuardForTea → **Talking** → GuardForCoffee
49. GuardForWater → GuardForTea → GuardForCoffee | **Talking**

Finally, "[Drinking]" consists in drinking only one of the drinks. There are three possibilities:

1. DrinkCoffee
2. DrinkTea
3. DrinkWater

As mentioned previously, we do not interpret in which situations these three possibilities are executed or not. We only know that one of them will be realized.

Overall, the number of possible scenarios, independently of the runtime data of the model, is thus: $6 + 7 - 1 + 49 + 3 = 64$.

B

Concurrency-aware Specification of fUML

Disclaimer: This concurrency-aware specification of fUML was made available alongside our SLE 2015 publication [85]. See http://gemoc.org/sle15/.

We show the different specifications constituting the concurrency-aware definition of fUML in the GEMOC Studio.

B.1 Abstract Syntax

The Abstract Syntax of fUML is shown as an Ecore metamodel on Figure 3.1.

B.2 Semantic Rules

The Semantic Rules of fUML were presented on Figure 3.5 as a metamodel extending the AS of fUML. Listing B.1 shows their full implementation using Kermeta 3 [32]. For technical reasons, the Execution Functions and Execution Data *must* be declared in the Abstract Syntax.

Listing B.1: Implementation, using Kermeta 3, of the Semantic Rules of fUML.

```
1  package org.gemoc.sample.fuml.dsa
2
3  import fr.inria.diverse.k3.al.annotationprocessor.Aspect
4  import java.util.HashMap
5  import java.util.List
6  import java.util.Map
7  import org.gemoc.sample.fuml.Action
8  import org.gemoc.sample.fuml.ActivityEdge
9  import org.gemoc.sample.fuml.ActivityNode
10 import org.gemoc.sample.fuml.ControlToken
11 import org.gemoc.sample.fuml.DecisionNode
12 import org.gemoc.sample.fuml.FinalNode
13 import org.gemoc.sample.fuml.ForkNode
14 import org.gemoc.sample.fuml.FumlFactory
15 import org.gemoc.sample.fuml.InitialNode
16 import org.gemoc.sample.fuml.JoinNode
17 import org.gemoc.sample.fuml.LiteralString
18 import org.gemoc.sample.fuml.MergeNode
19 import org.gemoc.sample.fuml.ObjectToken
20 import org.gemoc.sample.fuml.OpaqueAction
21 import org.gemoc.sample.fuml.OutputPin
22 import org.gemoc.sample.fuml.Token
23 import groovy.lang.GroovyShell
24 import groovy.lang.Script
25
26 // Semantic Rules (Execution Functions and Execution Data) of
      fUML
27 // Includes pre-conditions for each function to help debug the
      MoCMapping
28
29 @Aspect(className=ActivityNode)
30 class ActivityNodeAspect {
31
32  // Modifier
33  def public void execute() {
34   // Delete the tokens of incoming edges.
35   _self.incomingEdges.forEach [ incomingEdge |
36    incomingEdge.currentTokens.clear()
37   ]
38
39   // If an incomingEdge's source is a DecisionNode, clear the
      other outgoing edges of this decision node.
```

```
40    _self.incomingEdges.filter [ incomingEdge |
41     incomingEdge.sourceNode instanceof DecisionNode
42    ].map [ incomingEdge |
43     incomingEdge.sourceNode as DecisionNode
44    ].forEach [ decisionNode |
45     decisionNode.outgoingEdges.forEach [ outgoingEdge |
46      outgoingEdge.currentTokens.clear()
47     ]
48    ]
49   }
50
51   // Helper for debugging purposes.
52   def public void log() {
53    val sb = new StringBuilder()
54    sb.append("IncomingEdges: ")
55    _self.incomingEdges.forEach [ incomingEdge |
56     sb.append("[" + incomingEdge.name + ":" + incomingEdge.
      currentTokens + "]")
57    ]
58    sb.append("\n")
59    sb.append("OutgoingEdges: ")
60    _self.outgoingEdges.forEach [ outgoingEdge |
61     sb.append("[" + outgoingEdge.name + ":" + outgoingEdge.
      currentTokens + "]")
62    ]
63    if (_self instanceof Action
64     && !(_self as Action).outputs.isEmpty) {
65     sb.append("\n")
66     sb.append("OutputPins: ")
67     (_self as Action).outputs.forEach [ outputPin |
68      sb.append("[" + outputPin.name + ": "
69       + outputPin.currentTokens + "]")
70     ]
71    }
72    println(sb.toString)
73   }
74  }
75
76  @Aspect(className=InitialNode)
77  class InitialNodeAspect {
78   private boolean executed = false
79
80   // Make sure InitialNode is executed only once.
81   def private void precondition() {
```

```
82    if (!_self.executed) {
83      _self.executed = true
84    } else {
85      throw new PreconditionException(_self)
86    }
87  }
88
89  // Modifier
90  def public void execute() {
91    precondition(_self)
92    println("*** InitialNode [" + _self.name + "]" + "***")
93
94    // Create a ControlToken on each outgoing edge.
95    _self.outgoingEdges.forEach [ outgoingEdge |
96      outgoingEdge.currentTokens.add(
97        TokenHelper.createControlToken()
98      )
99    ]
100
101   ActivityNodeAspect.execute(_self)
102   // ActivityNodeAspect.log(_self)
103 }
104}
105
106@Aspect(className=FinalNode)
107class FinalNodeAspect {
108 private boolean executed = false
109
110 // Make sure FinalNode is executed only once.
111   def private void precondition() {
112     if (!_self.executed) {
113       _self.executed = true
114     } else {
115       throw new PreconditionException(_self)
116     }
117   }
118
119   // Modifier
120   def public void execute() {
121     precondition(_self)
122
123     println("*** FinalNode [" + _self.name + "]" + "***")
124
```

```
125     // Remove all tokens from incoming edges. If a token is an
        ObjectToken, print the object.
126     _self.incomingEdges.forEach [ incomingEdge |
127      incomingEdge.currentTokens
128       .filter(ObjectToken)
129       .forEach [ objectToken |
130        println("> " + objectToken.object.toString())
131       ]
132      incomingEdge.currentTokens.clear()
133     ]
134
135     ActivityNodeAspect.execute(_self)
136    // ActivityNodeAspect.log(_self)
137    }
138 }
139
140 @Aspect(className=MergeNode)
141 class MergeNodeAspect {
142
143    // Verify that at least one of the incoming edges has at least
        one token.
144    def private void precondition() {
145     val boolean atLeastOneIncomingEdgeHasAtLeastOneToken =
146      !_self.incomingEdges.map [ incomingEdge |
147       incomingEdge.currentTokens
148      ].flatten.isEmpty
149     if (!atLeastOneIncomingEdgeHasAtLeastOneToken) {
150       throw new PreconditionException(_self)
151     }
152    }
153
154    // Modifier
155    def public void execute() {
156     precondition(_self)
157
158     println("*** MergeNode [" + _self.name + "]" + "***")
159
160     // Transmits all incoming tokens to the outgoing edges.
161     val List<Token> incomingTokens =
162      _self.incomingEdges.map [ incomingEdge |
163       incomingEdge.currentTokens
164      ].flatten.toList
165
166     _self.outgoingEdges.forEach [ outgoingEdge |
```

```
167       outgoingEdge.currentTokens.addAll(incomingTokens)
168     ]
169     ActivityNodeAspect.execute(_self)
170   // ActivityNodeAspect.log(_self)
171   }
172 }
173
174 @Aspect(className=DecisionNode)
175 class DecisionNodeAspect {
176
177   // There should be at least one incoming edge with at least one
        token.
178   def private void precondition() {
179     val boolean atLeastOneIncomingEdgeHasAtLeastOneToken =
180       !_self.incomingEdges.map [ incomingEdge |
181         incomingEdge.currentTokens
182       ].flatten.isEmpty
183     if (!atLeastOneIncomingEdgeHasAtLeastOneToken) {
184       throw new PreconditionException(_self)
185     }
186   }
187
188   // Modifier
189   def public void execute() {
190     precondition(_self)
191
192     println("*** DecisionNode [" + _self.name + "]" + "***")
193
194     // Supposedly transmits all incoming tokens to the outgoing
        edges.
195     // For the purpose of this implementation, there should be
        only one incoming ObjectToken and it is passed to the outgoing
         edges so they can evaluate their guard.
196     // For animation purposes, the Execution Data must be in the
        AS so we have to take care when modifying lists we are
        iterating over.
197     if (_self.incomingEdges.size > 1) {
198       throw new UnsupportedOperationException(
199         "This implementation only supports DecisionNode with one
200           incoming edge. Found: " + _self.incomingEdges.size)
201     } else {
202       val incomingEdgeTokens =
203         _self.incomingEdges.get(0).currentTokens
204       if (incomingEdgeTokens.size > 1) {
```

```
205         throw new UnsupportedOperationException(
206           "This implementation only supports DecisionNode for one
207            token at a time. Found: " + incomingEdgeTokens.size)
208       } else {
209         val token = incomingEdgeTokens.get(0)
210         if (!(token instanceof ObjectToken)) {
211           throw new UnsupportedOperationException(
212            "This implementation only supports incoming ObjectTokens
213             for DecisionNode. Found: " + token)
214         } else {
215           _self.outgoingEdges.forEach [ outgoingEdge |
216            outgoingEdge.currentTokens.clear()
217            val Object object = (token as ObjectToken).object
218            outgoingEdge.currentTokens.add(
219             TokenHelper.createObjectToken(object)
220            )
221           ]
222         }
223       }
224     }
225
226     ActivityNodeAspect.execute(_self)
227   // ActivityNodeAspect.log(_self)
228   }
229 }
230
231 @Aspect(className=OpaqueAction)
232 class OpaqueActionAspect {
233
234   // There should be at least one incoming edge with at least one
       token.
235   def private void precondition() {
236     val boolean atLeastOneIncomingEdgeHasAtLeastOneToken =
237       !_self.incomingEdges.map [ incomingEdge |
238         incomingEdge.currentTokens
239       ].flatten.isEmpty
240     if (!atLeastOneIncomingEdgeHasAtLeastOneToken) {
241       throw new PreconditionException(_self)
242     }
243   }
244
245   // Modifier
246   def public void execute() {
247     precondition(_self)
```

```
248
249   println("*** OpaqueActionNode [" + _self.name + "]" + "***")
250
251   if (_self.bodies.size != _self.languages.size) {
252     throw new UnsupportedOperationException("OpaqueActions should
253      have the same number of bodies and languages.")
254   } else {
255     if (_self.bodies.size != 1) {
256      throw new UnsupportedOperationException("Only one body/
257       language supported in OpaqueAction.")
258     } else {
259      if (_self.languages.get(0) != "Groovy") {
260        throw new UnsupportedOperationException("Only Groovy is
261         supported as action language.")
262      } else {
263
264        // At this point, we know there is only one body and one
    language that is Groovy.
265        // First execute the body of the action using the Groovy
    interpreter.
266        val String body = _self.bodies.get(0)
267        val Object result = GroovySupport.execute(body, null)
268
269        // If Groovy returned some object, print it.
270        if (result != null) {
271         println(" => " + result)
272        }
273
274        // Create a ControlToken on every outgoing edge (if any).
275        _self.outgoingEdges.forEach [ outgoingEdge |
276         outgoingEdge.currentTokens.add(
277          TokenHelper.createControlToken()
278         )
279        ]
280
281        // Publish the result of the body of the action on the
    output pins.
282        _self.outputs.forEach [ outputPin |
283         val newToken = TokenHelper.createObjectToken(result)
284         outputPin.currentTokens.add(newToken)
285        ]
286
287        ActivityNodeAspect.execute(_self)
288      // ActivityNodeAspect.log(_self)
```

```
289         }
290       }
291     }
292   }
293 }
294
295 @Aspect(className=OutputPin)
296 class OutputPinAspect {
297
298   // There should be at least one incoming edge with at least one
      token.
299   def private void precondition() {
300     val boolean atLeastOneIncomingEdgeHasAtLeastOneToken =
301       !_self.incomingEdges.map [ incomingEdge |
302         incomingEdge.currentTokens
303       ].flatten.isEmpty
304     val boolean pinHasAtLeastOneToken = !_self.currentTokens.empty
305
306     if (!atLeastOneIncomingEdgeHasAtLeastOneToken
307       && !pinHasAtLeastOneToken) {
308         throw new PreconditionException(_self)
309     }
310   }
311
312   // Modifier
313   def public void execute() {
314     precondition(_self)
315
316     println("*** OutputPin [" + _self.name + "]" + "***")
317
318     // Transmits all currentTokens given by the owning Action to
      the outgoing edges.
319     _self.outgoingEdges.forEach [ outgoingEdge |
320       outgoingEdge.currentTokens.addAll(_self.currentTokens)
321     ]
322
323   // ActivityNodeAspect.log(  _self)
324   }
325 }
326
327 @Aspect(className=ActivityEdge)
328 class ActivityEdgeAspect {
329
```

```
330  // To evaluate a guard, we need at least one ObjectToken on the
       edge.
331  def private void precondition() {
332   if (!(_self.currentTokens.size == 1)) {
333     throw new PreconditionException(_self)
334   } else {
335     if (!(_self.currentTokens.get(0) instanceof ObjectToken)) {
336      throw new PreconditionException(_self)
337     }
338   }
339  }
340
341  // Query
342  def public boolean evaluateGuard() {
343   precondition(_self)
344
345   println("*** Evaluating Guard [" + _self.name + "]" + "****")
346   if (_self.guard == null) {
347     throw new NullPointerException("There is no guard
348      on this edge.")
349   } else {
350     if (!(_self.guard instanceof LiteralString)) {
351      throw new UnsupportedOperationException(
352        "Concrete type of ValueSpecification cannot be
353         dealt with yet: " + _self.guard)
354     } else {
355      val guard = _self.guard as LiteralString
356
357      val objects = _self.currentTokens
358       .filter[token|token instanceof ObjectToken]
359       .map[token|
360        token as ObjectToken]
361       .map[objectToken|objectToken.object]
362
363      if (objects.size != 1) {
364        throw new RuntimeException("There should only be one
365         object at this point.")
366      } else {
367        var boolean resultOfGuard
368        if (guard.value.equals("else")) {
369
370          // Default guard "else" in fUML always return true, but
     the branch is executed only if none of the other branches are
     possible.
```

```
371             resultOfGuard = true
372           } else {
373             val String object = objects.get(0) as String
374
375             // Compare the value specified in the guard to the value
        contained by the incoming ObjectToken.
376             resultOfGuard = guard.value.equals(object)
377           }
378           println("Guard [" + _self.name + "]
379             returned: " + resultOfGuard)
380           return resultOfGuard
381         }
382       }
383     }
384   }
385 }
386
387 @Aspect(className=ForkNode)
388 class ForkNodeAspect {
389
390   // There should be at least one incoming edge with at least one
        token.
391   def private void precondition() {
392     val boolean atLeastOneIncomingEdgeHasAtLeastOneToken =
393       !_self.incomingEdges.map [ incomingEdge |
394         incomingEdge.currentTokens
395       ].flatten.isEmpty
396     if (!atLeastOneIncomingEdgeHasAtLeastOneToken) {
397       throw new PreconditionException(_self)
398     }
399   }
400
401   // Modifier
402   def public void execute() {
403     precondition(_self)
404
405     println("*** ForkNode [" + _self.name + "]" + "***")
406
407     // Forks each incoming token and sends a version to each
        outgoing edge.
408     _self.outgoingEdges.forEach [ outgoingEdge |
409       outgoingEdge.currentTokens.clear()
410       _self.incomingEdges.forEach [ incomingEdge |
411         incomingEdge.currentTokens.forEach [ token |
```

```
412         if (token instanceof ObjectToken) {
413          val Object object = (token as ObjectToken).object
414          outgoingEdge.currentTokens.add(
415           TokenHelper.createObjectToken(object)
416          )
417         } else {
418          outgoingEdge.currentTokens.add(
419           TokenHelper.createControlToken()
420          )
421         }
422       ]
423      ]
424    ]
425    ActivityNodeAspect.execute(_self)
426   // ActivityNodeAspect.log(_self)
427   }
428 }
429
430 @Aspect(className=JoinNode)
431 class JoinNodeAspect {
432
433   // There should be at least one incoming edge with at least one
       token.
434   def private void precondition() {
435    val boolean allIncomingEdgesHaveAtLeastOneToken =
436     _self.incomingEdges.forall [ incomingEdge |
437      !incomingEdge.currentTokens.isEmpty
438     ]
439    if (!allIncomingEdgesHaveAtLeastOneToken) {
440      throw new PreconditionException(_self)
441    }
442   }
443
444   // Modifier
445   def public void execute() {
446    precondition(_self)
447
448    println("*** JoinNode [" + _self.name + "]" + "****")
449
450    // If all the tokens offered on the incoming edges are control
        tokens, then one control token is offered on the outgoing
       edge.
451    val allIncomingTokensAreControlTokens =
452     _self.incomingEdges.forall [ incomingEdge |
```

```
453       incomingEdge.currentTokens.forall [ incomingToken |
454         incomingToken instanceof ControlToken
455       ]
456     ]
457
458     if (allIncomingTokensAreControlTokens) {
459       // Only one outgoing edge for join nodes
460       _self.outgoingEdges.get(0).currentTokens.add(
461         TokenHelper.createControlToken()
462       )
463
464     } else {
465
466       // If some of the tokens offered on the incoming edges are
      control tokens and others are data tokens, then only the data
      tokens are offered on the outgoing edge. Tokens are offered on
       the outgoing edge in the same order they were offered to the
      join.
467       _self.incomingEdges.forEach [ incomingEdge |
468         incomingEdge.currentTokens.filter [ token |
469           token instanceof ObjectToken
470         ].forEach [ token |
471           _self.outgoingEdges.forEach [ outgoingEdge |
472             outgoingEdge.currentTokens.add(token)
473           ]
474         ]
475       ]
476     }
477     ActivityNodeAspect.execute(_self)
478   // ActivityNodeAspect.log(_self)
479   }
480 }
481
482 // Required to use the incoming token's objects as variables in
      the language used for the guard.
483 package class Context {
484   private Map<String, Object> environment = new HashMap()
485
486   def public void put(String name, Object value) {
487     this.environment.put(name, value)
488   }
489
490   def public Map<String, Object> getEnvironment() {
491     return this.environment
```

```
492   }
493 }
494
495 // Useful for debugging.
496 package class PreconditionException extends RuntimeException {
497   Object context
498
499   new(Object o) {
500    this.context = o
501   }
502
503 }
504
505 package class TokenHelper {
506   def package static ObjectToken createObjectToken(Object object) {
507    val ObjectToken res = FumlFactory.eINSTANCE.createObjectToken()
508    res.object = object
509    return res
510   }
511
512   def package static ControlToken createControlToken() {
513    return FumlFactory.eINSTANCE.createControlToken()
514   }
515 }
516
517 // Provides the means to execute an arbitrary string as a Groovy
      program.
518 class GroovySupport {
519   private static val shell = new GroovyShell()
520
521   //Parses the given string as a Groovy program and executes it.
      If there is a context, then it is set up before executing the
      program.
522   def public static Object execute(String program, Context context) {
523
524    // Parse the string to get a Groovy program.
525    val Script script = shell.parse(program)
526
527    // If there is a context, set it up.
528    if (context != null) {
529      context.environment.forEach [ name, value |
530       script.setProperty(name, value)
531      ]
532    }
```

```
533
534     // Run the script.
535     val Object result = script.run();
536
537     return result
538   }
539 }
```

B.3 Model of Concurrency Mapping

The concurrency-aware approach as described in Chapter 3 relies on the Event Structure MoC [160]. At the language level, the formalism is called EventType Structures. Listing B.2 shows the implementation, using the Event Constraint Language (ECL) [27], of the MoCMapping of fUML.

Listing B.2: Model of Concurrency Mapping of fUML defined using the Event Constraint Language (ECL).

```
1  -- Abstract Syntax (AS) of fUML.
2  import 'platform:/resource/org.gemoc.sample.fuml.model/model/fuml.
       ecore'
3
4  -- Core libraries of constraints of CCSL/MoccML
5  ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
       model/ccsllibrary/kernel.ccslLib"
6  ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
       model/ccsllibrary/CCSL.ccslLib"
7
8  -- Custom library of constraints, written in MoccML. AS-agnostic.
9  ECLimport "platform:/resource/org.gemoc.sample.fuml.mocc/lib/
       MyLibrary.moccml"
10
11
12 package fuml
13
14 -- MoCTriggers declaration.
15
16  context ActivityNode
17   -- Represents the execution of a node.
18   def: mocc_executeNode : Event = self
19
20  -- An edge can have its guard evaluated.
```

```
21  -- If it is outgoing of a DecisionNode, it may be allowed (
      mocc_mayExecuteTarget) or disallowed (mocc_mayNotExecuteTarget
      ) depending on the result of the guard.
22  -- If it is disallowed then it may not be executed (
      mocc_doNotExecuteTarget).
23  -- If it is allowed then it is either executed (
      mocc_doExecuteTarget) or it is not (mocc_doNotExecuteTarget).
24  context ActivityEdge
25    -- Represents the evaluation of the guard of an edge.
26    def: mocc_evaluateGuard : Event = self
27    -- Controlled Events for the Feedback Protocol.
28    def: mocc_mayExecuteTarget : Event = self
29    def: mocc_mayNotExecuteTarget : Event = self
30
31   -- Needed for local variable.
32   def: mocc_mayOrMayNotExecuteTarget : Event = self
33
34   -- Needed to differentiate allowed branches from executed
      branches.
35   def: mocc_doExecuteTarget : Event = self
36   def: mocc_doNotExecuteTarget : Event = self
37
38   -- Needed for local variable.
39   def: mocc_doOrDoNotExecuteTarget : Event = self
40
41   -- Required for technical reasons.
42  context DecisionNode
43   -- Needed for local variable.
44   def: mocc_unionOfAllMayExecuteWithNonDefaultGuard : Event = self
45   def: mocc_noOtherMayExecuteWithNonDefaultGuard : Event = self
46
47
48 -- Constraints defining the symbolic partial ordering.
49
50  context InitialNode
51   -- Execute InitialNode only once.
52   -- Since every node is executed before its outgoing edges'
      target, this will be the only node available for execution at
      start.
53   inv executeInitialNodeOnce:
54    let onlyOneFirst : Event = Expression OneTickAndNoMore(
55     self.mocc_executeNode
56    ) in
57     Relation Coincides(self.mocc_executeNode, onlyOneFirst)
```

```
58
59
60  context MergeNode
61   -- Execute Merge Node whenever any of the incoming edges'
      source is executed.
62   inv executeIfAnyIncomingEdgeSourceExecuted:
63    let anyIncomingEdgeSourceExecuted : Event =
64     Expression Union(self.incomingEdges.sourceNode.mocc_executeNode)
      in
65     Relation Precedes(anyIncomingEdgeSourceExecuted,
66      self.mocc_executeNode
67     )
68
69
70  context ActivityEdge
71   -- In general, execute the source before the target.
72   inv executeSourceBeforeTarget:
73    ((not (self.guard <> null))
74    and (not (self.targetNode.oclIsKindOf(MergeNode)))) implies(
75     Relation Precedes(self.sourceNode.mocc_executeNode,
76      self.targetNode.mocc_executeNode
77     )
78    )
79
80   -- Execute source of edge before evaluating the guard.
81   inv executeDecisionBeforeEvaluate:
82    ((self.guard <> null) implies (
83      Relation Alternates(self.sourceNode.mocc_executeNode,
84       self.mocc_evaluateGuard
85      )
86     )
87    )
88
89   -- Exclusive selection between may or may not following
      evaluation of the guard.
90   inv mayOrMayNotAfterEvaluate:
91    (self.guard <> null) implies (
92     Relation ExclusiveSelection(self.mocc_evaluateGuard,
93      self.mocc_mayExecuteTarget, self.mocc_mayNotExecuteTarget
94     )
95    )
96
97   -- Do and DoNot are exclusive and do can only happen after a "
      may" while "may not" implies a "do not".
```

```
98   inv doOrDoNotAfterMayOrMayNot:
99    (self.guard <> null) implies (
100    Relation SynchronousExclusionSubset(self.mocc_mayExecuteTarget,
101     self.mocc_mayNotExecuteTarget, self.mocc_doExecuteTarget,
102     self.mocc_doNotExecuteTarget
103    )
104   )
105
106  -- Do Execute means executing.
107  inv doExecuteMeansExecuting:
108   ((self.guard <> null) implies (
109     Relation Alternates(self.mocc_doExecuteTarget,
110      self.targetNode.mocc_executeNode
111     )
112    )
113   )
114
115  -- Dealing with the default guard 'else' is as follows:
116  -- It always returns true (mayExecute will always occur as a
      result of the feedback specification).
117  -- But it is executed only if none of the other branches may be
      executed.
118  inv doExecuteOfDefaultGuardOnlyPossibleIfNoOtherMayExecute:
119   (self.guard <> null) implies (
120    (self.guard.oclIsKindOf(LiteralString)) implies (
121     (self.guard.oclAsType(LiteralString).value = 'else') implies(
122      let noOtherMayExecuteWithNonDefaultGuard : Event =
123       self.sourceNode
124        .oclAsType(DecisionNode)
125        .mocc_noOtherMayExecuteWithNonDefaultGuard
126      in
127       Relation SubClock(self.mocc_doExecuteTarget,
128        noOtherMayExecuteWithNonDefaultGuard
129       )
130     )
131    )
132   )
133
134
135 context DecisionNode
136  -- Can only choose one of the outgoing branches.
137  inv doExecutesAreExclusive:
138   Relation Exclusion(self.outgoingEdges.mocc_doExecuteTarget)
139
```

```
140   -- All the evaluations of guards are coincident (does not
       matter, just more practical - can be used as Semantic
       Variation Point)
141   --  inv evaluatesCoincide:
142   --   Relation Coincides(self.outgoingEdges.mocc_evaluateGuard)
143
144   -- All the Feedback Consequences happen at the same time.
145   -- No additional constraint could provide information as to
       which nodes will be allowed to happen, therefore forcing
       synchronicity does not change anything at this point.
146   inv synchronousResponse:
147    Relation Coincides(self.outgoingEdges.mocc_mayOrMayNotExecuteTarget)
148
149   -- All the Feedback Consequences actions happen at the same
       time.
150   inv synchronousAction:
151    Relation Coincides(self.outgoingEdges.mocc_doOrDoNotExecuteTarget)
152
153   -- MayExecute means that DoExecute is possible.
154   inv mayExecuteMeansThatOneOfTheDoExecuteIsPossible:
155    let unionOfAllMayExecute : Event =
156     Expression Union(self.outgoingEdges.mocc_mayExecuteTarget)
157    in
158     let unionOfAllDoExecute : Event =
159      Expression Union(self.outgoingEdges.mocc_doExecuteTarget)
160     in
161      Relation Coincides(unionOfAllMayExecute, unionOfAllDoExecute)
162
163
164 context Action
165   -- If an Action has output pins, then execute them right after
       the node has executed.
166   inv executeOwnedPins:
167    (self.outputs->size() >0) implies(
168     let first : Event = self.outputs->first().mocc_executeNode in
169      Relation Alternates(self.mocc_executeNode, first)
170    )
171
172   -- All output pins are executed at the same time.
173   inv concurrentExecutionOfPins:
174    Relation Coincides(self.outputs.mocc_executeNode)
175
176
```

```
177 -- Constraints required for technical reason but are of little
      value to understand the MoCMapping of fUML.
178
179 context ActivityEdge
180  -- Kill some events when there is no guard.
181  inv killmayExecuteTargetIfThereIsNoGuard:
182   (not (self.guard <> null)) implies (
183    let zero1 : Integer = 0 in
184     let waitZero1 : Event =
185      Expression Wait(self.mocc_mayExecuteTarget, zero1)
186     in
187      Relation Coincides(self.mocc_mayExecuteTarget, waitZero1)
188   )
189
190  inv killmayNotExecuteTargetIfThereIsNoGuard:
191   (not (self.guard <> null)) implies (
192    let zero2 : Integer = 0 in
193     let waitZero2 : Event =
194      Expression Wait(self.mocc_mayNotExecuteTarget, zero2)
195     in
196      Relation Coincides(self.mocc_mayNotExecuteTarget, waitZero2)
197   )
198
199  inv killEvaluateIfThereIsNoGuard:
200   (not (self.guard <> null)) implies (
201    let zero3 : Integer = 0 in
202     let waitZero3 : Event =
203      Expression Wait(self.mocc_evaluateGuard, zero3)
204     in
205      Relation Coincides(self.mocc_evaluateGuard, waitZero3)
206   )
207
208  inv killdoExecuteIfThereIsNoGuard:
209   (not (self.guard <> null)) implies (
210    let zero4 : Integer = 0 in
211     let waitZero4 : Event =
212      Expression Wait(self.mocc_doExecuteTarget, zero4)
213     in
214      Relation Coincides(self.mocc_doExecuteTarget, waitZero4)
215   )
216
217  inv killdoNotExecuteIfThereIsNoGuard:
218   (not (self.guard <> null)) implies (
219    let zero5 : Integer = 0 in
```

```
220       let waitZero5 : Event =
221        Expression Wait(self.mocc_doNotExecuteTarget, zero5)
222       in
223        Relation Coincides(self.mocc_doNotExecuteTarget, waitZero5)
224     )
225
226   -- Definition
227   inv definitionMayExecuteOrMayNotExecute:
228    let mayOrMayNot : Event =
229     Expression Union(self.mocc_mayExecuteTarget,
230      self.mocc_mayNotExecuteTarget
231     ) in
232      Relation Coincides(
233       self.mocc_mayOrMayNotExecuteTarget,
234       mayOrMayNot
235      )
236
237   -- Definition
238   inv definitionDoExecuteOrDoNotExecute:
239    let doOrDoNot : Event =
240     Expression Union(
241      self.mocc_doExecuteTarget,
242      self.mocc_doNotExecuteTarget
243     ) in
244     Relation Coincides(self.mocc_doOrDoNotExecuteTarget, doOrDoNot)
245
246  context DecisionNode
247   -- Definition.
248   -- Gather into an event the "mayExecute" of all the branches
     with non-default guard.
249   inv unionOfAllMayExecuteWithNonDefaultGuardDefinition:
250    let atLeastOneOfTheEdgesWithNonDefaultGuardOccur : Event =
251     Expression Union(self.outgoingEdges
252      ->select(edge : ActivityEdge |
253       (edge).guard.oclIsKindOf(LiteralString)
254      )
255      ->select(edge : ActivityEdge |
256       (edge).guard.oclAsType(LiteralString).value <> 'else'
257      ).mocc_mayExecuteTarget
258     )
259     in Relation Coincides(
260      self.mocc_unionOfAllMayExecuteWithNonDefaultGuard,
261      atLeastOneOfTheEdgesWithNonDefaultGuardOccur
262     )
```

```
263
264   -- Definition.
265   inv noOtherMayExecuteDefinition:
266    Relation Exclusion(self.mocc_noOtherMayExecuteWithNonDefaultGuard,
267     self.mocc_unionOfAllMayExecuteWithNonDefaultGuard
268    )
269
270   -- Without this it could occur in other steps. This is only for
       after a decision node's outgoing branches' guards are
      evaluated.
271   inv noOtherMayExecuteDefinitionContext1:
272    let unionOfAllDoExecute1 : Event =
273     Expression Union(self.outgoingEdges.mocc_doExecuteTarget)
274    in
275     Relation SubClock(self.mocc_noOtherMayExecuteWithNonDefaultGuard,
276      unionOfAllDoExecute1
277     )
278
279   inv noOtherMayExecuteDefinitionContext2:
280    let unionOfAllDoExecute2 : Event =
281     Expression Union(self.outgoingEdges.mocc_doExecuteTarget)
282    in
283     Relation SubClock(
284      self.mocc_unionOfAllMayExecuteWithNonDefaultGuard,
285      unionOfAllDoExecute2
286     )
287
288   inv anotherMayOrNoOtherMayCoincidesWithMaysAndDos:
289    let anotherMayOrNoOtherMay : Event =
290     Expression Union(
291      self.mocc_unionOfAllMayExecuteWithNonDefaultGuard,
292      self.mocc_noOtherMayExecuteWithNonDefaultGuard
293     ) in
294     let unionOfAllDos : Event =
295      Expression Union(self.outgoingEdges.mocc_doExecuteTarget)
296     in
297      Relation Coincides(anotherMayOrNoOtherMay, unionOfAllDos)
298
299 endpackage
```

In this listing, we have made use of a MoCCML library for the relations `ExclusiveSelection` and `SynchronousExclusionSubset`. We could have defined these relations alongside the MoCMapping, but we have implemented them as a relations

library which is shown on Listing B.3. As described in Section 3.11, they are domain-agnostic and can thus be used for the MoCMapping of any xDSML. This library is imported at the top of Listing B.2.

Listing B.3: Library of MoCCML relations used by the MoCMapping of fUML.

```
1  AutomataConstraintLibrary myLibrary {
2     import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
      model/ccsllibrary/kernel.ccslLib" as kernel ;
3     import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
      model/ccsllibrary/CCSL.ccslLib" as CCSLLib ;
4
5     RelationLibrary relations {
6        // Implementation of the Relation ExclusiveSelection
7        // Alternation between an EventType mapped to a query and
      its result.
8        // Its result is either May or MayNot, depending on the
      runtime state of the model thanks to the Feedback Protocol.
9        RelationDefinition ExclusiveSelectionDef[ExclusiveSelection]{
10          Relation Exclusion(Clock1 -> may, Clock2 -> mayNot)
11          Expression result = Union(Clock1 -> may, Clock2 -> mayNot)
12          Relation Alternates(
13           AlternatesLeftClock -> query,
14           AlternatesRightClock -> result
15          )
16       }
17
18       // Implementation of the Relation
      SynchronousExclusionSubset
19       // MayDo and MayNotDo are exclusive; so are DoIt and DoNot.
20       // If MayDo occurs, then DoIt or DoNot occurs.
21       // If MayNoDot occurs, then DoNot occurs.
22       RelationDefinition SynchronousExclusionSubsetDef
23        [SynchronousExclusionSubset]{
24          Expression mayOrMayNot =
25           Union(Clock1 -> mayDo, Clock2 -> mayNotDo)
26          Expression doOrDoNot =
27           Union(Clock1 -> doIt, Clock2 -> doNot)
28          Relation Exclusion(Clock1 -> doIt, Clock2 -> doNot)
29          Relation Exclusion(Clock1 -> mayDo, Clock2 -> mayNotDo)
30          Relation SubClock(LeftClock -> doIt, RightClock -> mayDo)
31          Relation SubClock(LeftClock -> doNot,
32           RightClock -> mayOrMayNot
33          )
```

```
34            Relation Coincides(Clock1 -> mayOrMayNot,
35             Clock2 -> doOrDoNot
36            )
37        }
38
39        // Declaration of the Relation ExclusiveSelection
40        RelationDeclaration ExclusiveSelection(
41         query : clock,
42         may : clock,
43         mayNot : clock
44        )
45
46        // Declaration of the Relation SynchronousExclusionSubset
47        RelationDeclaration SynchronousExclusionSubset(
48         mayDo : clock,
49         mayNotDo : clock,
50         doIt : clock,
51         doNot: clock
52        )
53    }
54 }
```

B.4 Communication Protocol

The Communication Protocol of fUML, defined using GEL, is shown on Listing 3.36.

C

Graphical Animation of the Example fUML Model During its Execution

Disclaimer: a video of the model execution described below was uploaded alongside our SLE 2015 publication [85]. See http://gemoc.org/sle15/.

Figure C.1 shows the example fUML Activity. This figure is based on the graphical concrete syntax we have devised for fUML, using Sirius[1]. Alongside this graphical concrete syntax, we have defined the animation layer of the language, *i.e.*, how the Execution Data should be represented on the graphical syntax during the execution of the model.

Our animation layer for fUML is composed of 2 decorations (green or yellow "play" symbols added over nodes to show that they can be executed, respectively that they may conditionally be executed) and 3 style customizations (to display the tokens held by edges and pins at runtime).

Figure C.2 shows the initial view of the Modeling Workbench when launching the execution of the example fUML Activity.

There are four elements in this figure. In the top left corner (Figure C.3) is the graphical representation of the Activity, including the animation layer (the green "play" symbol).

[1] http://www.eclipse.org/sirius/

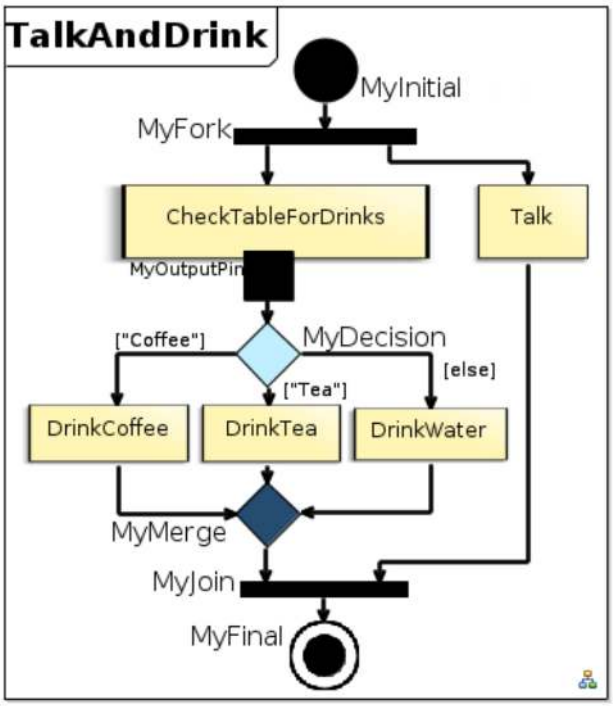

Figure C.1: Example fUML activity where we want to drink something from the table while talking.

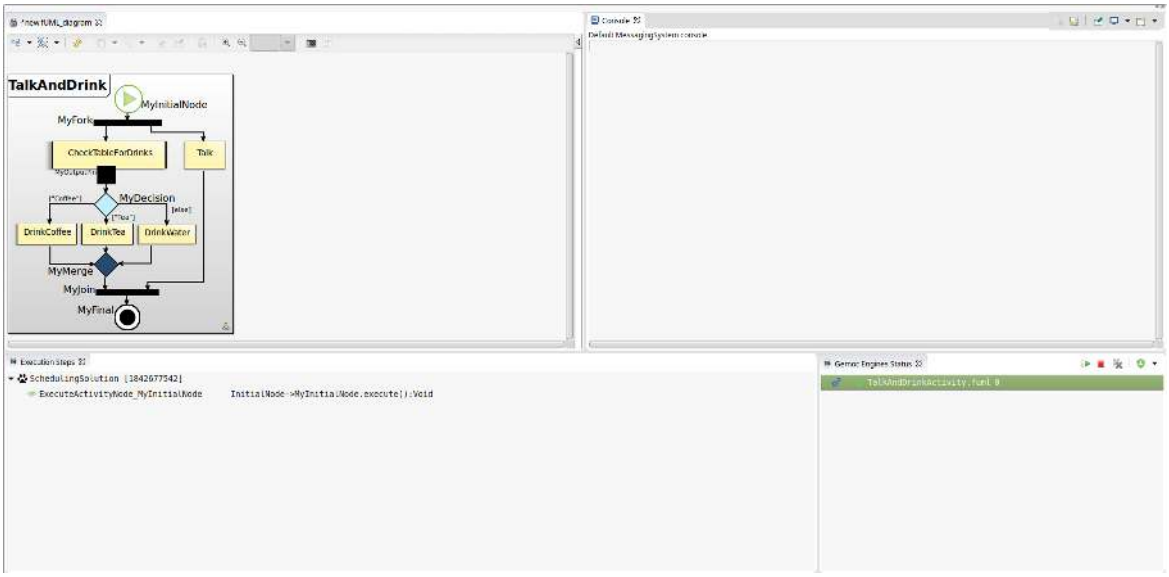

Figure C.2: Step 0 – Global view of the Modeling Workbench at the beginning of the execution of the example fUML Activity.

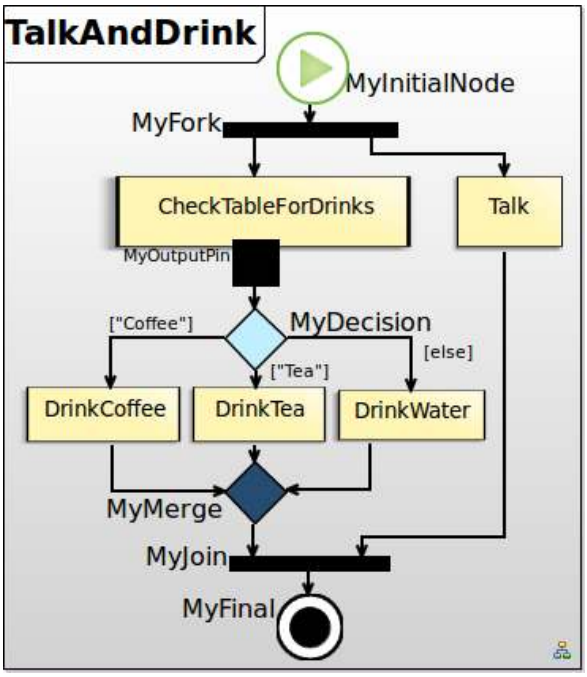

Figure C.3: Step 0 – Graphical animation of the example fUML Activity.

In the top right corner is the console, which is used as the standard text output by the Semantic Rules. It is particularly useful when developing or debugging a language or its animation layer. It can also be useful to print some information which is not suited for graphical representation.

In the bottom left corner is the heuristic of the Execution Engine (Figure C.4). It presents the possible Scheduling Solutions, along with the corresponding MappingApplications and Execution Function Calls. The user can realize choices at runtime in order to guide the execution, choices which would have otherwise been realize at random by the runtime. All proposed choices respect the semantics of the language. In the beginning, the only possible execution step consists in executing `MyInitialNode.execute()`.

Figure C.4: Step 0 – Default heuristic of the Execution Engine, presenting the possible execution steps to the user through a Graphical User Interface.

Finally, in the bottom right corner (Figure C.5) is the Execution Engine registry which allows us to control the selected Execution Engine, for instance to stop the execution.

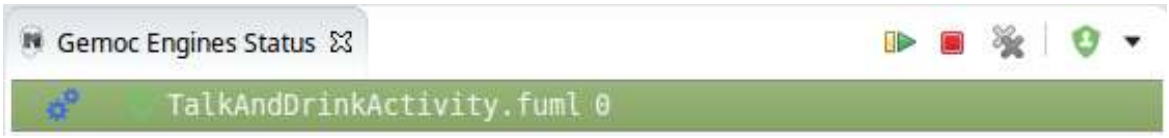

Figure C.5: Step 0 – The GEMOC Execution Engine registry.

Let us select the only available execution step: it triggers the execution of the InitialNode "MyInitialNode". Figure C.6 shows the view after selecting the execution step.

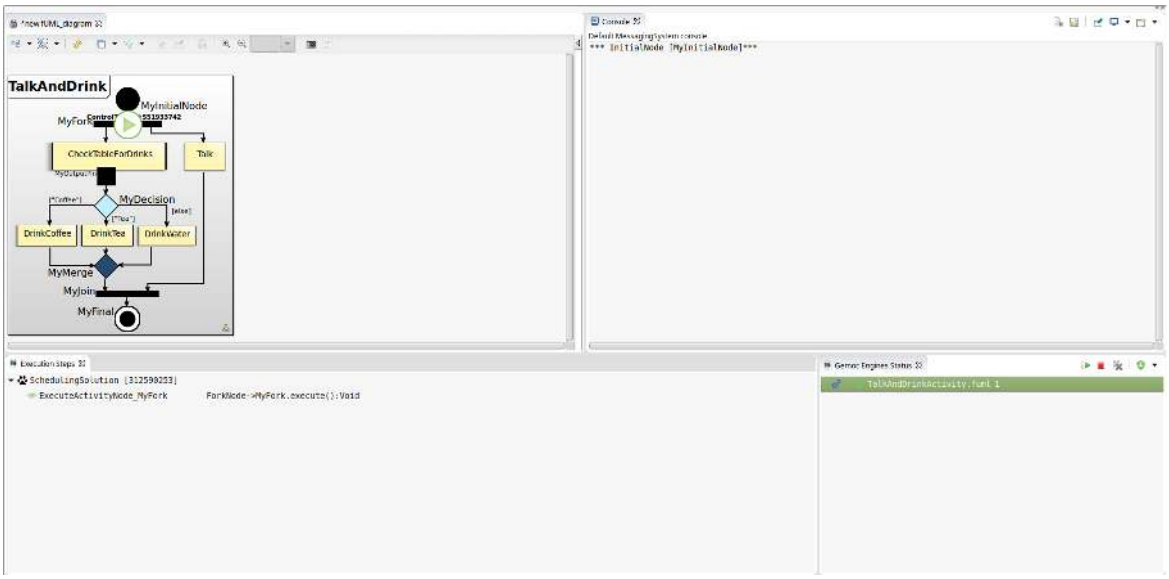

Figure C.6: Step 1 – Global view of the Modeling Workbench after the first step of execution.

- The model has been updated due to the execution of the InitialNode. Therefore, a Token has been created on the edge between "MyInitialNode" and "MyFork".
- The heuristic has been updated, the next possible execution step is to execute the ForkNode "MyFork".
- The execution of `MyInitialNode.execute()` has sent some text to be printed on the standard output console.
- In the engine registry, we can see that the engine is now at step 1 when it was previously at step 0.

Figure C.7 shows the model after selecting the execution step consisting in executing the ForkNode “MyFork”.

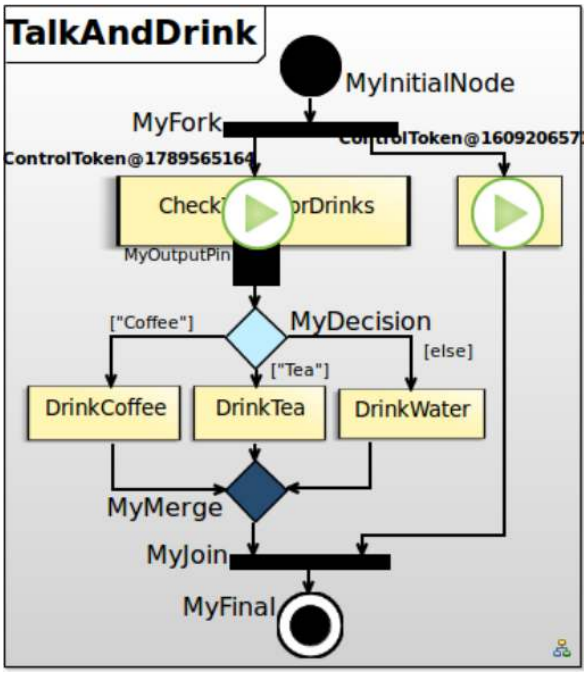

Figure C.7: Step 2 – Graphical animation of the example fUML Activity.

At this point, there are several solutions, as illustrated by the execution steps presented by the heuristic of the engine (Figure C.8). We can first execute the “CheckTableForDrinks”

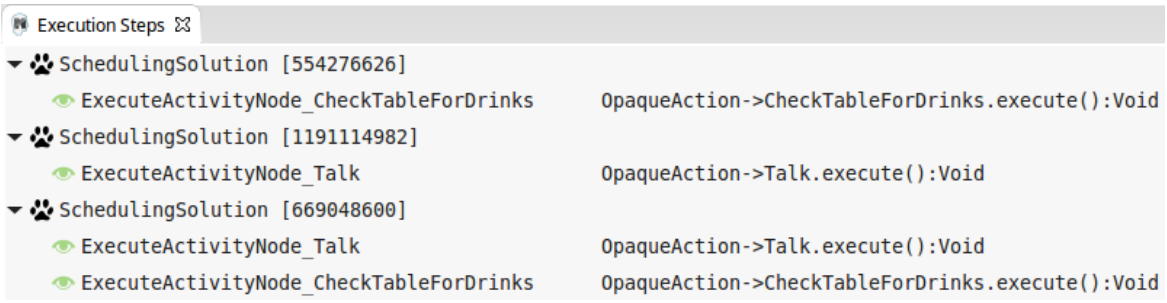

Figure C.8: Step 2 – Heuristic of the execution engine after executing the ForkNode.

node, first execute the “Talk” node, or do both in parallel. Ultimately, both branches must be executed, and the order in which it is done does not matter with regards to the semantics of fUML.

We select the last execution step, which contains occurrences of both mappings. The result is as shown on Figure C.9. The “CheckTableForDrinks” node has returned “Tea”.

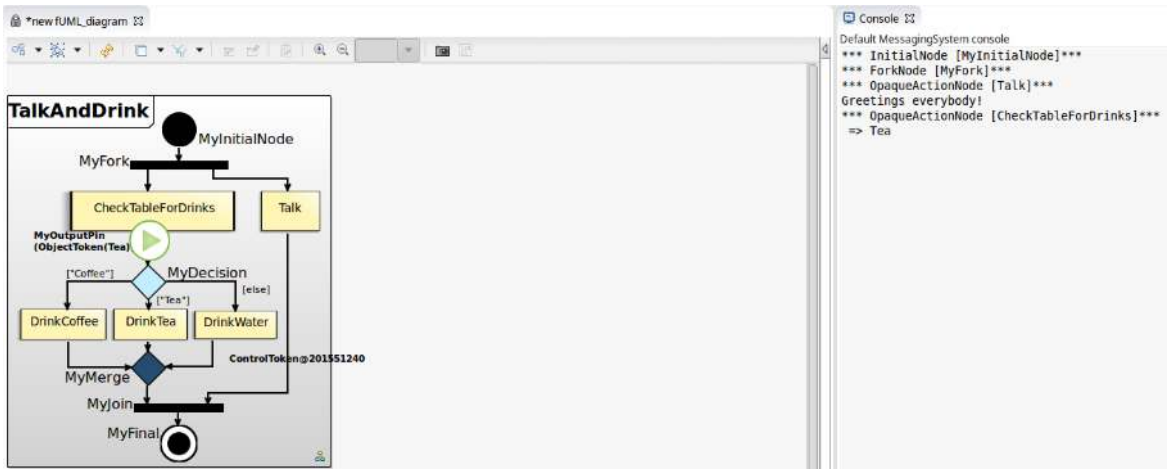

Figure C.9: Step 3 – Graphical animation of the example fUML Activity and standard output console during the execution.

Let us continue executing the rest of the branch corresponding to the drinking part of the activity. Figure C.10 shows the model after executing the OutputPin of "CheckTableForDrinks".

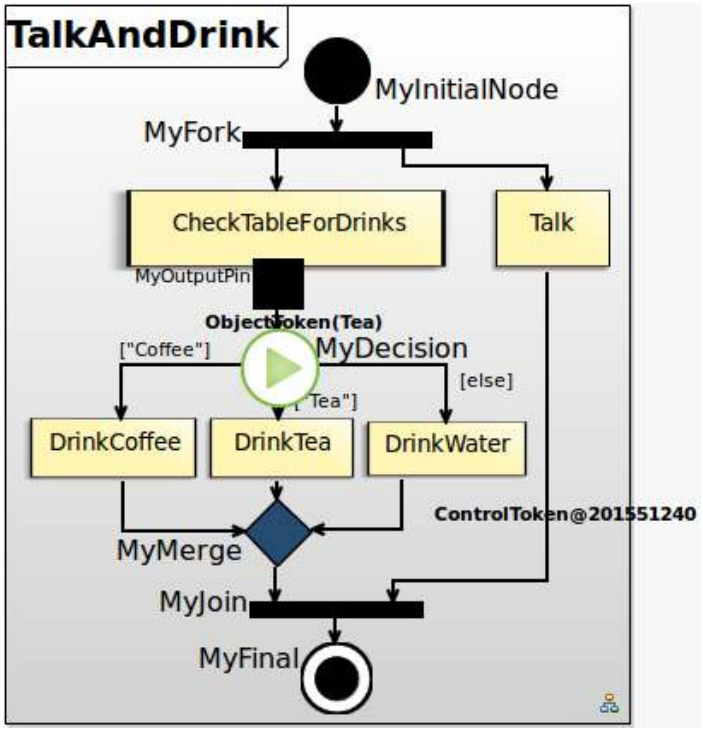

Figure C.10: Step 4 – Graphical animation of the example fUML Activity.

Figure C.11 shows the model and the possible execution steps after the DecisionNode "MyDecision" has been executed. As explained in details in Appendix A, the guards can be evaluated in any order, even possibly in parallel. Therefore, many execution steps are possible. Ultimately, all the guards must be executed.

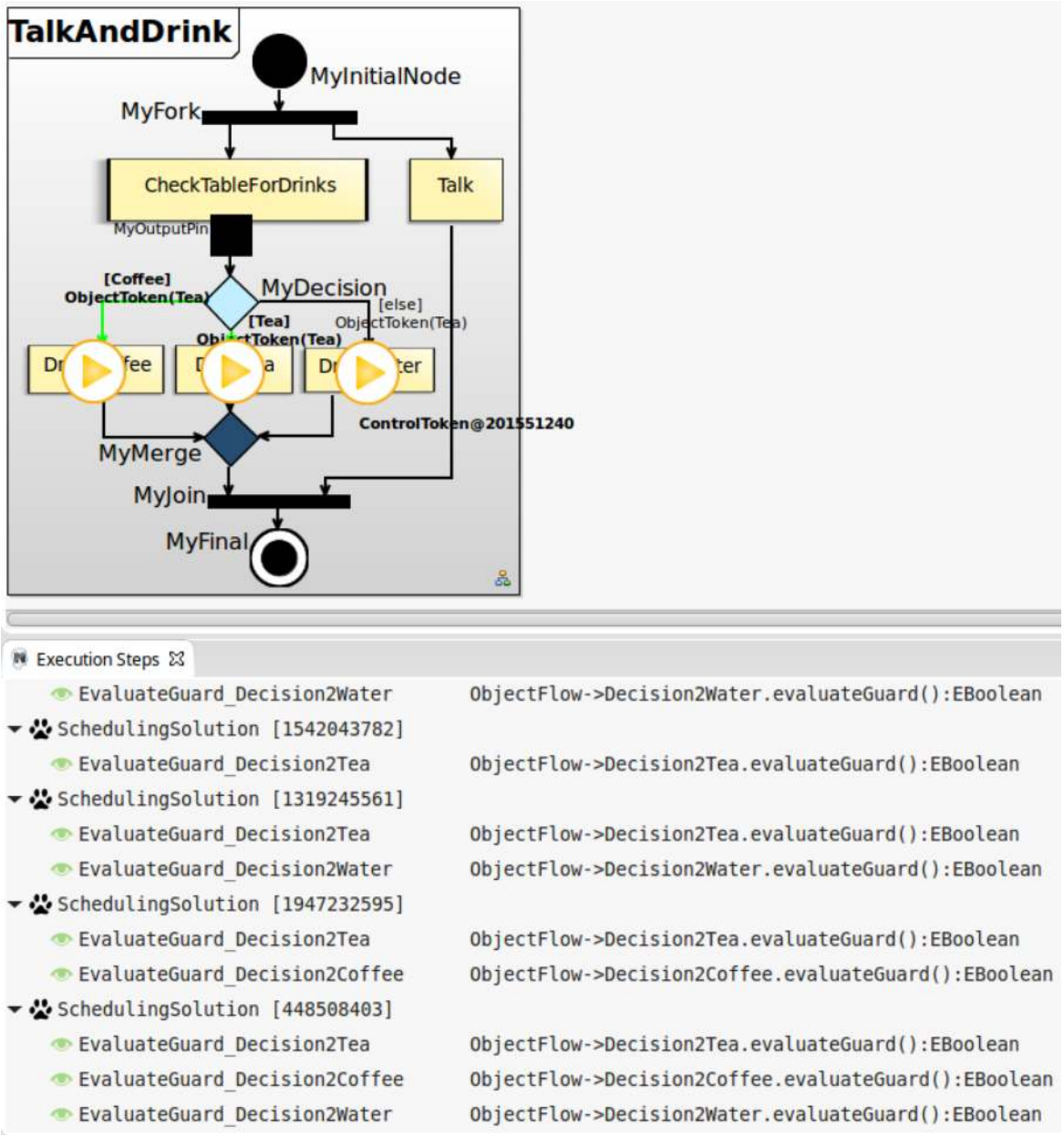

Figure C.11: Step 5 – Graphical animation of the example fUML Activity and heuristic of the execution engine showing all the possibilities in scheduling the evaluation of the guards.

Let us evaluate all the guards in one step. For each guard, its evaluation consists in checking if the value it specifies corresponds to the value on the incoming token. Figure C.12 shows the text sent to the standard output by the evaluation of the guards. These

```
*** Evaluating Guard [Decision2Tea]***
Guard [Decision2Tea] returned: true
*** Evaluating Guard [Decision2Water]***
Guard [Decision2Water] returned: true
*** Evaluating Guard [Decision2Coffee]***
Guard [Decision2Coffee] returned: false
```

Figure C.12: Step 6 – Text sent to the standard output console by the queries evaluating the guards outside the DecisionNode.

Execution Function calls are queries which return a boolean value. As explained in Section 3.6, this boolean value is then interpreted by a corresponding Feedback Policy. As a consequence, some of the future execution steps are forbidden from being selected because they are not consistent with regards to the runtime state of the model (represented by the returned boolean values).

Therefore, there is only one possible execution step afterwards, corresponding to drinking tea, as shown on Figure C.13.

Figure C.13: Step 7 – Only one execution step is allowed as a consequence of the results of the guards and of the application of the Feedback Policy (*cf.* Section 3.6).

The MergeNode “MyMerge” can then be executed since one of the branches of the DecisionNode has been executed. Figure C.14 shows this possibility.

When both branches of the ForkNode have finished executing (*i.e.*, once the MergeNode and the Talk node have both been executed), the JoinNode “MyJoin” may be executed, as shown on Figure C.14.

Finally, Figure C.16 shows that the FinalNode “MyFinal” may be executed to complete the Activity.

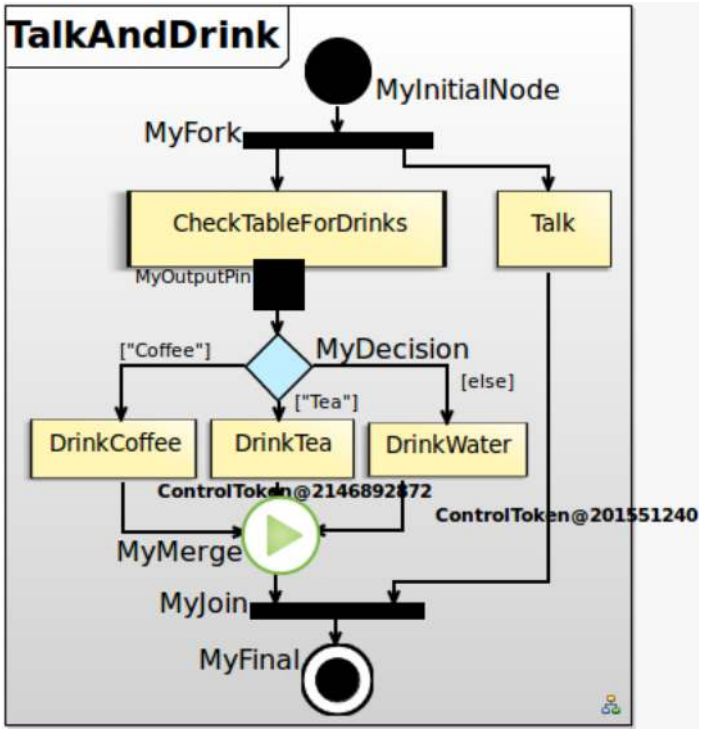

Figure C.14: Step 8 – Graphical animation of the example fUML Activity.

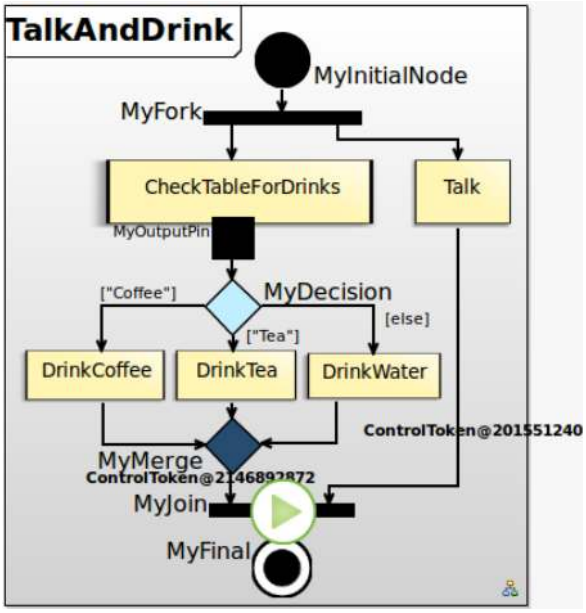

Figure C.15: Step 9 – Graphical animation of the example fUML Activity.

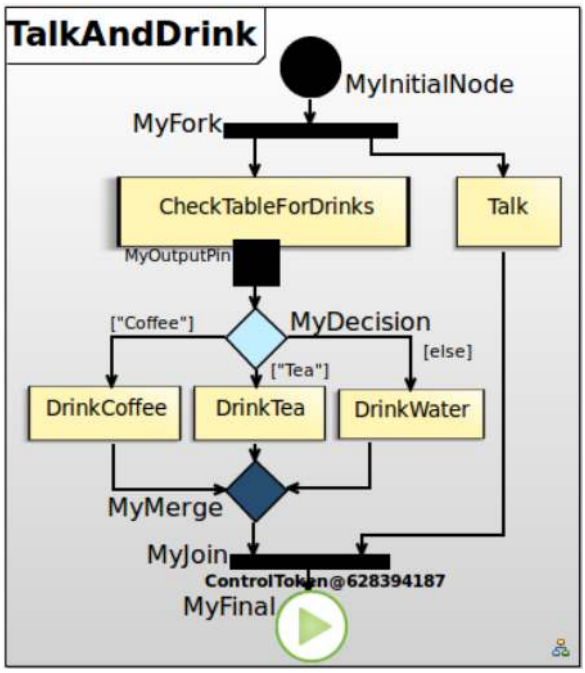

Figure C.16: Step 10 – Graphical animation of the example fUML Activity.

D

Concurrency-aware Specification of the Threading xDSML

We show the different specifications constituting the concurrency-aware definition of the Threading xDSML in the GEMOC Studio.

D.1 Abstract Syntax

Figure 4.6 showed an excerpt from the abstract syntax of the Threading xDSML. Figure D.1 shows the whole abstract syntax, as an Ecore metamodel, of our implementation of this language.

A ThreadSystem is composed of Threads, including a main one. Threads consist of Tasks which may be of different natures. An ExecutionTask is a basic task. A Conditional has a set of conditions which must all be true before its "then" clause is executed. Otherwise, if it has an "else" clause, it is executed. A disjunction has a set of operands, one of which is executed. A Task may start a Thread (StartThreadTask) or wait for one to finish executing (JoinThreadTask). Finally, a ProxyTask is used to represent another Task so that the same Task can be referenced at several points in the program. Each Thread is scheduled according to an Agenda, consisting of Instructions which represent Tasks.

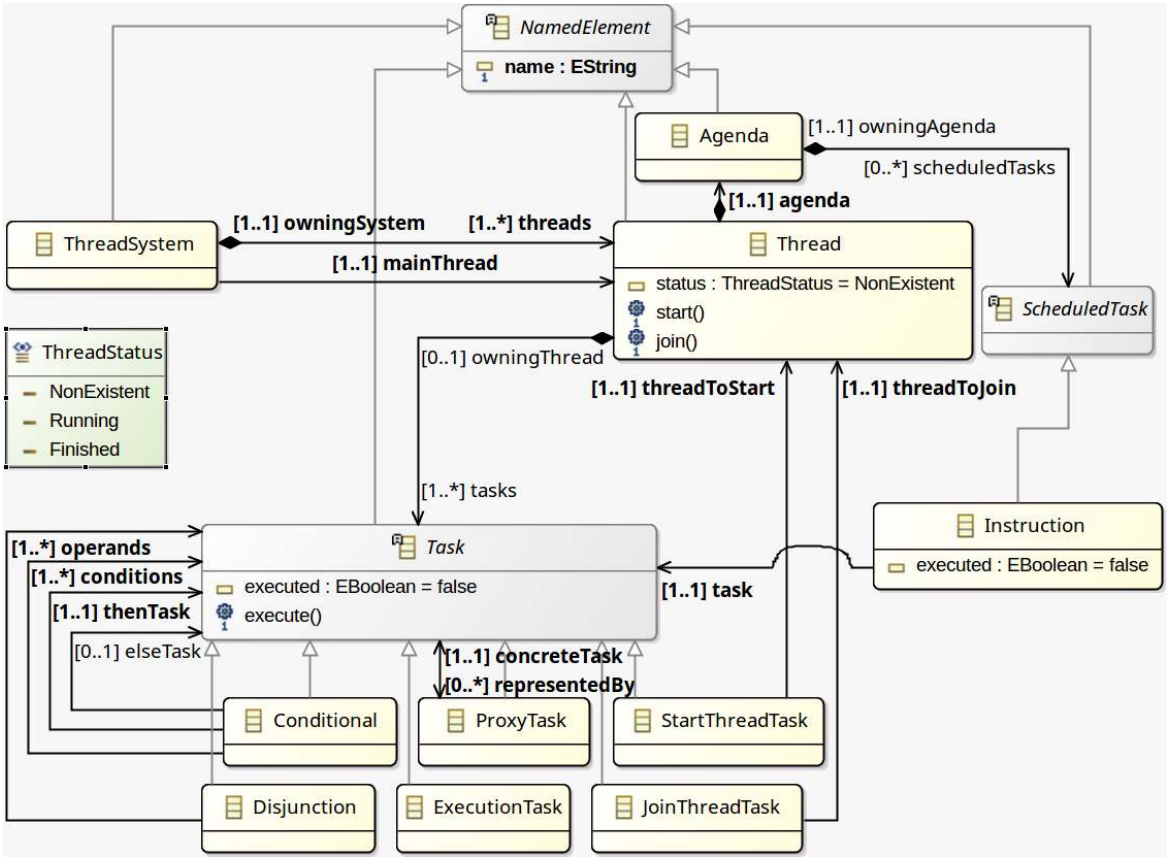

Figure D.1: Metamodel representing the abstract syntax of the Threading xDSML.

D.2 Semantic Rules

For technical reasons, the Execution Functions and Data must be declared in the metamodel representing the abstract syntax of the language. Their implementation, however, is realized using Kermeta 3, as shown on Listing D.1.

Listing D.1: Excerpt from the Semantic Rules of fUML specified using Kermeta 3.

```
1  package org.gemoc.sample.threaded.dsa
2
3  import fr.inria.diverse.k3.al.annotationprocessor.Aspect
4  import org.gemoc.sample.threaded.Conditional
5  import org.gemoc.sample.threaded.Disjunction
6  import org.gemoc.sample.threaded.ExecutionTask
7  import org.gemoc.sample.threaded.Instruction
8  import org.gemoc.sample.threaded.JoinThreadTask
9  import org.gemoc.sample.threaded.NamedElement
10 import org.gemoc.sample.threaded.ScheduledTask
```

```
11 import org.gemoc.sample.threaded.StartThreadTask
12 import org.gemoc.sample.threaded.Task
13 import org.gemoc.sample.threaded.Thread
14 import org.gemoc.sample.threaded.ThreadStatus
15 import org.gemoc.sample.threaded.ThreadSystem
16
17 @Aspect(className=typeof(Thread))
18 class ThreadAspect extends NamedElementAspect {
19
20  // Modifier
21  // Starts the Thread
22  def public void start() {
23   println("Starting Thread: " + _self.name)
24   _self.status = ThreadStatus.RUNNING
25  }
26
27  // Modifier
28  // Terminates the Thread
29  def public void join() {
30   println("Joining Thread: " + _self.name)
31   _self.status = ThreadStatus.FINISHED
32  }
33 }
34
35 @Aspect(className=typeof(Task))
36 abstract class TaskAspect extends NamedElementAspect {
37
38  // Modifier
39  // Executes the Task.
40  // Also sets the Instruction that triggered the Task as executed
      .
41  def public void execute() {
42   println("Executing Task: " + _self.name)
43
44   _self.executed = true
45
46   val instructions =
47    _self.owningThread.agenda.scheduledTasks
48     .filter [ scheduledTask |
49     scheduledTask instanceof Instruction
50    ].map[ scheduledTask |
51     scheduledTask as Instruction
52    ]
53   val instruction = instructions
```

```
54      .findFirst [ instruction |
55       instruction.task.equals(_self)
56       || ((instruction.task instanceof Disjunction)
57        && (instruction.task as Disjunction).operands.contains(_self)
         )
58      || ((instruction.task instanceof Conditional)
59       && (instruction.task as Conditional).thenTask.equals(_self))
60      ]
61
62    if (instruction != null) {
63     instruction.executed = true
64    }
65   }
66  }
```

D.3 Model of Concurrency Mapping

The concurrency-aware approach as described in Chapter 3 relies on the Event Structure MoC [160]. At the language level, the formalism is called EventType Structures. Listing D.2 shows the implementation, using the Event Constraint Language (ECL) [27], of the MoCMapping of the Threading xDSML.

Listing D.2: Model of Concurrency Mapping of the Threading xDSML defined using the Event Constraint Language (ECL).

```
1  import 'platform:/resource/org.gemoc.sample.threaded.model/model/
       threaded.ecore'
2
3  ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
       model/ccsllibrary/kernel.ccslLib"
4  ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.
       model/ccsllibrary/CCSL.ccslLib"
5
6  package threaded
7
8  -- MoCTriggers of the EventType Structure.
9
10  context Thread
11   -- Represents the start and the end of a Thread's execution.
12   def: mocc_start : Event = self
13   def: mocc_join : Event = self
14
```

```
15  context Task
16   -- Represents the execution of a Task
17   def: mocc_execute : Event = self
18
19  context Conditional
20   -- Represents the execution of the 'then' clause.
21   def: mocc_executeThenTask : Event = self
22   -- Represents the execution of nothing, if there is no 'else'
      clause.
23   def: mocc_doNothing : Event = self
24   -- Represent whether all the conditions were executed or not
25   def: mocc_conditionsWereOk : Event = self
26   def: mocc_conditionsAreOk : Event = self
27
28  context ScheduledTask
29    -- Represents the execution of an Instruction in the Agenda.
30   def: mocc_occur : Event = self
31
32
33 -- Constraints defining the symbolic partial ordering.
34
35  context Thread
36   -- For every start of a Thread, there is a join.
37   inv alternateStartAndJoin:
38    Relation Alternates(self.mocc_start, self.mocc_join)
39
40   -- For every start of a Thread, its first scheduled task occurs
      .
41   inv alternateStartAndFirstTask:
42    Relation Alternates(
43     self.mocc_start,
44     self.agenda.scheduledTasks->first().mocc_occur
45    )
46
47   -- For every occurrence of its last scheduled task, a Thread
      joins.
48   inv alternateLastTaskAndJoin:
49    Relation Alternates(
50     self.agenda.scheduledTasks->last().mocc_occur,
51     self.mocc_join
52    )
53
54
55  context Instruction
```

```
56  -- An Instruction only occurs if the previous scheduled tasks
     of the agenda have occurred.
57  inv occurOnlyAfterPreviousTasks:
58   (self.owningAgenda <> null
59    and self.owningAgenda.scheduledTasks->first() <> self
60   ) implies
61    let supOfPreviousScheduledTasks : Event =
62     Expression Sup(
63      self.owningAgenda.scheduledTasks->select( scheduledTask |
64       self.owningAgenda.scheduledTasks->indexOf(scheduledTask)
65        < self.owningAgenda.scheduledTasks->indexOf(self)
66      ).mocc_occur
67     )
68   in
69     Relation Alternates(supOfPreviousScheduledTasks, self.mocc_occur)
70
71  -- An Instruction must occur before the scheduled tasks which
     are later in the agenda.
72  inv executeBeforeNextTasks:
73   (self.owningAgenda <> null
74    and self.owningAgenda.scheduledTasks->last() <> self
75   ) implies
76    let infOfNextScheduledTasks : Event =
77     Expression Inf(
78      self.owningAgenda.scheduledTasks->select(scheduledTask |
79       self.owningAgenda.scheduledTasks->indexOf(scheduledTask)
80        > self.owningAgenda.scheduledTasks->indexOf(self)
81      ).mocc_occur
82     )
83    in
84     Relation Alternates(self.mocc_occur, infOfNextScheduledTasks)
85
86  -- If the executed Task is a "JoinThreadTask" then the current
     Thread must block on the other Thread's join.
87  inv forJoinTasksTheOwningThreadMustWait
88   ForTheThreadToJoinToHavefinished:
89   (self.owningAgenda <> null
90    and self.owningAgenda.scheduledTasks->last() <> self
91    and self.task.oclIsKindOf(JoinThreadTask)
92   ) implies
93    let infOfNextScheduledTasks2 : Event =
94     Expression Inf(
95      self.owningAgenda.scheduledTasks->select(scheduledTask |
96       self.owningAgenda.scheduledTasks->indexOf(scheduledTask)
```

```
97          > self.owningAgenda.scheduledTasks->indexOf(self)
98        ).mocc_occur
99       )
100     in
101      Relation Alternates(
102       self.task.oclAsType(JoinThreadTask).threadToJoin.mocc_join,
103       infOfNextScheduledTasks2
104      )
105
106   -- When an Instruction occurs, it means that its corresponding
      task is executed.
107   inv occurrenceMeansExecutingTask:
108    Relation Coincides(self.mocc_occur, self.task.mocc_execute)
109
110   -- If the underlying Task is a Conditional, then the
      Conditional's 'then' branch is executed or it is not, before
      the next scheduled tasks in the agenda can proceed.
111   inv executeOneBranchOfTheConditional:
112    (self.owningAgenda <> null
113     and self.owningAgenda.scheduledTasks->last() <> self
114      and self.task.oclIsKindOf(Conditional)
115    ) implies
116     let eitherBranchExecution : Event =
117      Expression Union(
118       self.task.oclAsType(Conditional).thenTask.mocc_execute,
119        self.task.oclAsType(Conditional).mocc_doNothing
120      ) in
121       let infOfNextScheduledTasks3 : Event =
122        Expression Inf(self.owningAgenda.scheduledTasks
123         ->select(scheduledTask |
124          self.owningAgenda.scheduledTasks->indexOf(scheduledTask)
125           > self.owningAgenda.scheduledTasks->indexOf(self)
126         ).mocc_occur
127       ) in
128        Relation Alternates(
129         eitherBranchExecution,
130         infOfNextScheduledTasks3
131        )
132
133
134 context StartThreadTask
135   -- A Task of type "StartThreadTask" means that the
      corresponding Thread must be started.
136   inv executeMeansStartingThread:
```

```
137     Relation Coincides(self.mocc_execute, self.threadToStart.mocc_start)
138
139
140  context Disjunction
141   -- In a Disjunction, only one of the operands is executed.
142   inv onlyOneTaskExecuted:
143    Relation Exclusion(self.operands.mocc_execute)
144
145   -- When either of its operands is executed, it means that the
      Disjunction has occurred.
146   inv occurWhenOneTaskExecutes:
147    let unionOfOperandsExecution : Event =
148     Expression Union(self.operands.mocc_execute)
149    in
150     Relation Coincides(unionOfOperandsExecution, self.mocc_execute)
151
152
153  context Task
154   -- If some proxies represent this task, then this task must be
      executed whenever one of the proxies is executed.
155   inv ifHasProxyThenExecuteWhenOneOfTheProxiesIsExecuted:
156    (self.representedBy->notEmpty()) implies
157     let unionOfProxiesExecute : Event =
158      Expression Union(self.representedBy.mocc_execute)
159     in
160      Relation Coincides(unionOfProxiesExecute, self.mocc_execute)
161
162  context Conditional
163   -- Condition is validated when all the conditions
164   -- have executed.
165   inv conditionsOkDef :
166    let supOfConditions : Event =
167     Expression Sup(self.conditions.mocc_execute)
168    in
169     Relation Coincides(self.mocc_conditionsWereOk, supOfConditions)
170
171   -- If the conditions were OK then
172   -- we must execute the 'thenTask' sometime.
173   inv executeWhenConditionsHaveExecuted:
174    Relation Alternates(
175     self.mocc_conditionsWereOk,
176     self.mocc_executeThenTask
177    )
178
```

```
179   -- When conditions are OK we execute the 'thenTask'.
180   inv executeThenWhenConditionsAreOk:
181    Relation Coincides(
182     self.mocc_conditionsAreOk,
183     self.mocc_executeThenTask
184    )
185
186   -- Conditions are OK if they were OK previously and this
       Conditional is being executed.
187   inv conditionsAreOkDef:
188    let supOfConditionsWereOkAndSelfExecutes : Event =
189     Expression Sup(self.mocc_conditionsWereOk, self.mocc_execute)
190    in
191     Relation Coincides(
192      self.mocc_conditionsAreOk,
193      supOfConditionsWereOkAndSelfExecutes
194     )
195
196   -- If this conditional's 'thenTask' must be executed, then we
       execute the then task.
197   inv executeThenTaskDef:
198    Relation Alternates(
199     self.mocc_executeThenTask,
200     self.thenTask.mocc_execute
201    )
202
203   -- Either the 'thenTask' of the Conditional is executed, or it
       is not.
204   inv resultAnyway:
205    let unionOfExecuteAndNotExecute : Event =
206     Expression Union(self.mocc_executeThenTask, self.mocc_doNothing)
207    in
208      Relation Coincides(unionOfExecuteAndNotExecute, self.mocc_execute)
209
210   -- We can't have both the 'thenTask' execute and do nothing.
211   inv exclusionOfResult:
212    Relation Exclusion(self.mocc_executeThenTask, self.mocc_doNothing)
213
214 endpackage
```

D.4 Communication Protocol

The Communication Protocol is specified using the GEMOC Events Language (GEL) described in Chapter 3.

Listing D.3: Excerpt from the textual concrete syntax of GEL.

```
1  import "platform:/plugin/org.gemoc.sample.threaded.model/model/
       threaded.ecore"
2  import "platform:/plugin/org.gemoc.sample.threaded.mocc/ECL/
       Threaded.ecl"
3
4  DSE StartThread:
5    upon mocc_start
6    triggers Thread.start blocking
7  end
8
9  DSE JoinThread:
10   upon mocc_join
11   triggers Thread.join blocking
12 end
13
14 DSE ExecuteTask:
15   upon mocc_execute
16   triggers Task.execute blocking
17 end
18
19 DSE ExecuteInstruction:
20   upon mocc_occur
21 end
```

E

Concurrency-aware Specification of fUML Using the Threading xDSML as MoC

We detail the concurrency-aware definition of fUML in the GEMOC Studio, using the Threading xDSML as MoC. This xDSML was introduced in Subsection 4.2.2 and its implementation is presented in Appendix D.

E.1 Abstract Syntax

The fUML abstract syntax is provided as an Ecore metamodel on Figure 3.1.

E.2 Semantic Rules

The Semantic Rules are given in Xtend in Appendix B.

E.3 Model of Concurrency Mapping

The MoCMapping, in the recursive definition of concurrency-aware xDSML we have proposed in Chapter 4, is implemented by an Abstract Syntax Transformation between the

syntax of the domain (fUML) and the syntax of the MoC (Threading). Listing E.1 shows the full source code, in Xtend [7], of that transformation. As explained in Chapter 4, this transformation also generates the model-level Projections.

Disclaimer: so far, this implementation does not support nested Fork/Join pairs, as it was not needed for the example fUML Activity. The last method should be completed for it to work.

Listing E.1: The MoCMapping of fUML using the Threading xDSML as MoC, specified using Xtend.

```
1  package org.gemoc.sample.fuml.mapping.threaded
2
3  import com.google.common.collect.Sets
4  import java.util.ArrayList
5  import java.util.Collection
6  import java.util.Comparator
7  import java.util.HashMap
8  import java.util.HashSet
9  import java.util.LinkedList
10 import java.util.List
11 import java.util.Map
12 import java.util.Queue
13 import java.util.Set
14 import org.eclipse.emf.ecore.resource.Resource
15 import org.gemoc.gel.projections.LanguageProjection
16 import org.gemoc.gel.projections.Projections
17 import org.gemoc.gel.projections.ProjectionsFactory
18 import org.gemoc.sample.fuml.Action
19 import org.gemoc.sample.fuml.Activity
20 import org.gemoc.sample.fuml.ActivityEdge
21 import org.gemoc.sample.fuml.ActivityNode
22 import org.gemoc.sample.fuml.DecisionNode
23 import org.gemoc.sample.fuml.ForkNode
24 import org.gemoc.sample.fuml.JoinNode
25 import org.gemoc.sample.fuml.LiteralString
26 import org.gemoc.sample.fuml.OutputPin
27 import org.gemoc.sample.threaded.Thread
28 import org.gemoc.sample.threaded.ThreadSystem
29 import org.gemoc.sample.threaded.ThreadedFactory
30 import org.gemoc.sample.threaded.ProxyTask
31 import org.gemoc.sample.threaded.Task
32 import org.gemoc.sample.fuml.MergeNode
33
```

```
34 // Model transformation of fUML towards the Threading language,
       used as its Model of Concurrency.
35 // Corresponds to the Model of Concurrency Mapping for fUML.
36
37 class Fuml2ThreadedMapping {
38   val threadedFactory = ThreadedFactory.eINSTANCE
39   val projectionsFactory = ProjectionsFactory.eINSTANCE
40   val Collection<LanguageProjection> languageProjections
41   val Projections modelProjections
42   val Collection<Task> tasksNotNecessarilyExecuted
43    = new ArrayList()
44
45   new(Resource languageProjectionsResource,
46       Resource modelProjectionsResource) {
47     // Load the language projections
48     val projections =
49       languageProjectionsResource.contents.get(0) as Projections
50     this.languageProjections = projections.languageProjections
51
52     // Create the model projections
53     val modelProjections = projectionsFactory.createProjections
54     modelProjectionsResource.contents.add(modelProjections)
55     this.modelProjections = modelProjections
56   }
57
58 // Entry point of the transformation.
59   def void perform(Resource fumlResource,
60       Resource threadedResource) {
61     val activity = fumlResource.contents.get(0) as Activity
62
63     // Transform the fUML Activity
64     val threadSystem = transform(activity)
65     threadedResource.contents.add(threadSystem)
66
67     // Add the necessary imports into the model projections
68     val importFumlModel = projectionsFactory.createImportStatement
69     importFumlModel.importURI =
70       fumlResource.URI.toPlatformString(true)
71     modelProjections.imports.add(importFumlModel)
72     val importThreadedModel =
73       projectionsFactory.createImportStatement
74     importThreadedModel.importURI =
75       threadedResource.URI.toPlatformString(true)
76     modelProjections.imports.add(importThreadedModel)
```

```
77  }
78
79  // Transforms an fUML Activity into a Threading model.
80  def ThreadSystem transform(Activity activity) {
81
82    // Create the root object
83    val threadSystem = threadedFactory.createThreadSystem
84    threadSystem.name = activity.name + "ThreadSystem"
85
86    // There is at least one thread per activity.
87    val mainThread = threadedFactory.createThread
88    mainThread.name = "mainThreadFor" + threadSystem.name
89    threadSystem.threads.add(mainThread)
90    threadSystem.mainThread = mainThread
91
92    // For each pair of Fork/Join, create a thread for every
      branch.
93    // Identify pairs of Fork/Join
94    val Collection<Pair<ForkNode, JoinNode>> pairsOfForkAndJoin =
95      findNodePairs(activity, ForkNode, JoinNode)
96    val Collection<Pair<DecisionNode, MergeNode>>
      pairsOfDecisionAndMerge =
97      findNodePairs(activity, DecisionNode, MergeNode)
98
99    pairsOfForkAndJoin.forEach [ pair |
100     println(pair.key.name + " --- " + pair.value.name)
101   ]
102
103   // Identify branches for each pair, with which nodes are on
      it.
104   val Map<ActivityNode, Pair<ForkNode, ActivityEdge>>
      nodeLocations =
105     findBranchesHoldingNodes(activity, pairsOfForkAndJoin)
106
107   //  For each branch on which there are nodes, create a Thread
      and add it to the ThreadSystem.
108   val mapOfBranchToThread = new HashMap()
109   val branches = nodeLocations.values.toSet
110   branches.forEach [ pair |
111     val newThread = threadedFactory.createThread
112     newThread.name =
113       "Thread" + "_" + pair.key.name + "_" + pair.value.name
114     mapOfBranchToThread.put(pair, newThread)
115     threadSystem.threads.add(newThread)
```

```
116     ]
117
118     // We know which nodes are on which branches, we want to find
        which nodes are on the main thread.
119     val allNodes = new ArrayList(activity.nodes)
120     allNodes.addAll(
121       activity.nodes.filter[node|node instanceof Action]
122           .map[node|(node as Action).outputs]
123           .flatten
124      )
125
126     val nodesOnMainThread =
127       allNodes.filter[node|!(nodeLocations.keySet.contains(node))]
128
129     // The nodes should be in the right order to ensure the
      instructions will be in the right order.
130     val branchesAndTheirContents =
131       new HashMap<
132         Pair<ForkNode, ActivityEdge>, List<ActivityNode>
133       >
134     branches.forEach [ branch |
135       branchesAndTheirContents.put(branch,
136         nodeLocations.entrySet
137           .filter[entry|entry.value == branch]
138           .map[entry|entry.key].toList)
139     ]
140
141     val Comparator<ActivityNode> comparator = new Comparator<
      ActivityNode>() {
142       override compare(ActivityNode o1, ActivityNode o2) {
143         // Return -1 when o1 is before o2, that is when there is
      a trail of edges from o1 to o2
144         // Return 1 if the contrary
145         // Otherwise return 0
146         if (nodeIsAncestorOf(o2, o1)) {
147           return 1
148         } else if (nodeIsAncestorOf(o1, o2)) {
149           return -1
150         } else {
151           return 0
152         }
153       }
154     }
155
```

```
156     val sortedNodesOnMainThread =
157       nodesOnMainThread.sortWith(comparator)
158     val sortedBranchesAndTheirContents =
159       branchesAndTheirContents.mapValues [ listOfNodes |
160         listOfNodes.sortWith(comparator)
161       ]
162
163     // We now have:
164     // 1 - One main thread and as many threads as branches
165     // 2 - Which nodes are on the main threads
166     // 3 - Which nodes are on each branch
167     // We only need to create the Task corresponding to each node
        on the correct thread.
168     // If we use ProxyTasks we need to link afterwards
169     val mapOfProxies = new HashMap()
170     sortedNodesOnMainThread.forEach [ node |
171       mapOfProxies.putAll(
172         toTask(node, mainThread, mapOfBranchToThread,
    pairsOfForkAndJoin)
173       )
174     ]
175     sortedBranchesAndTheirContents.entrySet.forEach [ entry |
176       entry.value.forEach [ node |
177         mapOfProxies.putAll(
178           toTask(node, mapOfBranchToThread.get(entry.key),
179             mapOfBranchToThread, pairsOfForkAndJoin)
180           )
181       ]
182     ]
183
184     // Each Proxy, when created, has specified as a result of
    which ActivityNode it was born.
185     // When every Task has been created, we can retrieve the Task
        corresponding to the wanted Node.
186     mapOfProxies.entrySet.forEach [ entry |
187       val node = entry.value
188       val proxyTask = entry.key
189       val taskCorrespondingToNode =
190         modelProjections.modelProjections.findFirst [ projection |
191           projection.name.contains("ForExecution") && projection.
    languageElement == node
192         ].moccElement as Task
193       proxyTask.concreteTask = taskCorrespondingToNode
194     ]
```

```
195
196     // If a node is on a branch of a Decision Node, it is not
        necessarily executed
197     allNodes
198     .filter[node |
199       pairsOfDecisionAndMerge.exists[pair |
200       nodeIsAncestorOf(pair.key, node)
201         && nodeIsAncestorOf(node, pair.value)
202       ]
203     ].map[node |
204       modelProjections.modelProjections.filter[projection |
205         projection.name.contains("ForExecution")
206       ].findFirst[projection |
207         projection.languageElement == node
208       ].moccElement as Task
209     ].forEach [ task |
210       tasksNotNecessarilyExecuted.add(task)
211     ]
212
213     // Create the Agenda for each Thread
214     threadSystem.threads.forEach[ thread |
215       createAgenda(thread, tasksNotNecessarilyExecuted)
216     ]
217
218     return threadSystem
219   }
220
221   private def void createAgenda(Thread thread,
222             Collection<Task> tasksToRemove) {
223     val agenda = threadedFactory.createAgenda
224     agenda.name = "Agenda_" + thread.name
225     val instructions = new ArrayList(thread.tasks)
226       .filter[task|! tasksToRemove.contains(task)]
227       .map [ task |
228         val instruction = threadedFactory.createInstruction
229         instruction.name = task.name
230         instruction.owningAgenda = agenda
231         instruction.task = task
232         instruction
233     ]
234     agenda.scheduledTasks.addAll(instructions)
235     thread.agenda = agenda
236   }
237
```

```
238  // Transforms the given ActivityNode into a (set of) Task on
       the given thread. May use the map of threads to reference the
       correct one (i.e. in case of Fork/Join)
239  private def Map<ProxyTask, ActivityNode> toTask(
240   ActivityNode node, Thread thread,
241   Map<Pair<ForkNode, ActivityEdge>, Thread> mapOfBranchToThread,
242   Collection<Pair<ForkNode, JoinNode>> pairsOfForkAndJoin) {
243
244    // Used later on to do the linking part for ProxyTasks.
245    val result = new HashMap()
246
247    // Retrieve the different Projections of fUML
248    // When transforming a Node, we will create a ModelProjection
       corresponding to what we did.
249    val LanguageProjection projectionForEvaluation =
250      this.languageProjections.findFirst [ projection |
251        projection.name.contains("ForEvaluation")
252      ]
253    val LanguageProjection projectionForExecution =
254      this.languageProjections.findFirst [ projection |
255        projection.name.contains("ForExecution")
256      ]
257    val LanguageProjection projectionForMayExecute =
258      this.languageProjections.findFirst [ projection |
259        projection.name.contains("ForMayExecute")
260      ]
261    val LanguageProjection projectionForMayNotExecute =
262      this.languageProjections.findFirst [ projection |
263        projection.name.contains("ForMayNotExecute")
264      ]
265
266    switch (node) {
267      ForkNode case true: {
268        // Create as many "Start Thread" tasks as there are
       branches on this ForkNode.
269        val tasksCreatedForNode = new ArrayList()
270        node.outgoingEdges.forEach [ edge |
271          val branch = new Pair(node, edge)
272          val threadToStart = mapOfBranchToThread.get(branch)
273          val task = threadedFactory.createStartThreadTask
274          task.name = "StartThread_" + threadToStart.name
275          task.owningThread = thread
276          task.threadToStart = threadToStart
277          tasksCreatedForNode.add(new Pair(task, node))
```

```
278         ]
279
280         // Create the ModelProjection corresponding to this
    transformation
281         // We decide that the *first* StartThread Task resulting
    of a ForkNode transformation is the one that is mapped to the
    ForkNode's execution.
282         val lastTaskForNode = tasksCreatedForNode.head
283         val modelProjection = projectionsFactory.
    createModelProjection
284         modelProjection.name =
285           projectionForExecution.name + "_" +
286             lastTaskForNode.value.name
287         modelProjection.moccElement = lastTaskForNode.key
288         modelProjection.languageElement = lastTaskForNode.value
289         modelProjections.modelProjections.add(modelProjection)
290       }
291
292       JoinNode case true: {
293         // Create as many "Join Thread" tasks as there are
    branches incoming to this JoinNode.
294         val tasksCreatedForNode = new ArrayList()
295         val associatedForkNode =
296           pairsOfForkAndJoin.findFirst[pair|pair.value == node].key
297         associatedForkNode.outgoingEdges.forEach [ edge |
298           val branch = new Pair(associatedForkNode, edge)
299           val threadToJoin = mapOfBranchToThread.get(branch)
300           val task = threadedFactory.createJoinThreadTask
301           task.name = "JoinThread_" + threadToJoin.name
302           task.owningThread = thread
303           task.threadToJoin = threadToJoin
304           tasksCreatedForNode.add(new Pair(task, node))
305         ]
306
307         // Create the ModelProjection corresponding to this
    transformation
308         // We decide that the *last* JoinThread Task resulting of
     a JoinNode transformation is the one that is mapped to the
    JoinNode's execution.
309         val lastTaskForNode = tasksCreatedForNode.last
310         val modelProjection =
311           projectionsFactory.createModelProjection
312         modelProjection.name =
313           projectionForExecution.name + "_"
```

```
314             + lastTaskForNode.value.name
315         modelProjection.moccElement = lastTaskForNode.key
316         modelProjection.languageElement = lastTaskForNode.value
317         modelProjections.modelProjections.add(modelProjection)
318       }
319
320       DecisionNode case true: {
321         // Create a task for the Decision itself
322         val task = threadedFactory.createExecutionTask
323         task.name = "Execution_" + node.name
324         task.owningThread = thread
325         // Create the corresponding ModelProjection
326         val modelProjectionForDecisionNode =
327           projectionsFactory.createModelProjection
328         modelProjectionForDecisionNode.name =
329           projectionForExecution.name + "_" + node.name
330         modelProjectionForDecisionNode.moccElement = task
331         modelProjectionForDecisionNode.languageElement = node
332         modelProjections.modelProjections.add(
333           modelProjectionForDecisionNode
334         )
335
336         // Transform the decision node into a sequence of (
    evaluate guard of a branch, disjunction of its result)
337         // followed by a sequence of conditionals capturing all
    possible outcomes.
338         node.outgoingEdges.forEach [ edge |
339           // First create a Task for the evaluation of the guard
340           val taskEvaluate = threadedFactory.createExecutionTask
341           taskEvaluate.name =
342             "Execute_" + edge.name + "_EvaluateGuard"
343           taskEvaluate.owningThread = thread
344           // And the corresponding ModelProjection
345           val modelProjection =
346             projectionsFactory.createModelProjection
347           modelProjection.name =
348             projectionForEvaluation.name + "_" + edge.name
349           modelProjection.moccElement = taskEvaluate
350           modelProjection.languageElement = edge
351           modelProjections.modelProjections.add(modelProjection)
352           // Then create the disjunction between its two outcomes
353           val taskMay = threadedFactory.createExecutionTask
354           taskMay.name =
355             "ExecutionTask_May" + edge.targetNode.name
```

```
356             taskMay.owningThread = thread
357             tasksNotNecessarilyExecuted.add(taskMay)
358             val taskMayNot = threadedFactory.createExecutionTask
359             taskMayNot.name =
360               "ExecutionTask_MayNot" + edge.targetNode.name
361             taskMayNot.owningThread = thread
362             tasksNotNecessarilyExecuted.add(taskMayNot)
363             val disjunctionTask = threadedFactory.createDisjunction
364             disjunctionTask.name =
365               "Disjunction_MayOrMayNot" + edge.targetNode.name
366             disjunctionTask.owningThread = thread
367             disjunctionTask.operands.add(taskMay)
368             disjunctionTask.operands.add(taskMayNot)
369             // And the corresponding ModelProjections
370             val modelProjectionMay =
371               projectionsFactory.createModelProjection
372             modelProjectionMay.name =
373               projectionForMayExecute.name + "_" + edge.name
374             modelProjectionMay.moccElement = taskMay
375             modelProjectionMay.languageElement = edge
376             modelProjections.modelProjections.add(
377               modelProjectionMay
378             )
379             val modelProjectionMayNot =
380               projectionsFactory.createModelProjection
381             modelProjectionMayNot.name =
382               projectionForMayNotExecute.name + "_" + edge.name
383             modelProjectionMayNot.moccElement = taskMayNot
384             modelProjectionMayNot.languageElement = edge
385             modelProjections.modelProjections.add(
386               modelProjectionMayNot
387             )
388           ]
389
390           // Now we want to create a sequence of conditionals
391           // First create the set of all possible outcomes
392           val Set<Set<ActivityEdge>> possiblePermutations =
393             computePossiblePermutations(node.outgoingEdges)
394
395           // Order them by the number of elements
396           val orderedPermutations =
397             possiblePermutations
398             .sortBy[set|set.size]
399             .reverse
```

```
400
401         // Create the tasks corresponding to all the possible
    sets.
402         // First a conditional with the MayExecute of every edge
    of the set
403         // If there is more than 1 solution, create a disjunction
404         // In any case, target some ProxyTask created for the
    purpose of this conditional
405         orderedPermutations.forEach [ set |
406           val conditionalTask = threadedFactory.createConditional
407           conditionalTask.name =
408             set.join("Conditional_", "_", "_Allowed",
409               [edge|edge.name]
410             )
411           conditionalTask.owningThread = thread
412           val mayTasksForEdges = set.map [ edge |
413             modelProjections.modelProjections
414             .findFirst [ projection |
415               projection.name.contains(
416                 projectionForMayExecute.name
417               )
418                 && projection.languageElement == edge
419             ].moccElement as Task
420           ]
421           conditionalTask.conditions.addAll(mayTasksForEdges)
422           if (set.size == 1
423             || (set.size == 2
424               && set.exists[edge|edgeHasDefaultGuard(edge)]
425             )
426           ) {
427             val edgeToConsider = if (set.size == 1) {
428                 set.get(0)
429               } else {
430                 set.findFirst[edge|! edgeHasDefaultGuard(edge)]
431               }
432             val thenTask = threadedFactory.createProxyTask
433             thenTask.name = "Proxy" +
434               (result.values.filter[activityNode|
435                 activityNode == edgeToConsider.targetNode
436                 ].toList.size + 1
437               ) + "For"
438               + edgeToConsider.targetNode.name
439             thenTask.owningThread = thread
440             result.put(thenTask, edgeToConsider.targetNode)
```

```
441             conditionalTask.thenTask = thenTask
442             tasksNotNecessarilyExecuted.add(thenTask)
443           } else {
444             // Only consider the non-default branches
445             val nonDefaultBranches =
446               set.filter[edge|! edgeHasDefaultGuard(edge)]
447             val thenTask = threadedFactory.createDisjunction
448             thenTask.name = "Disjunction_"
449               + nonDefaultBranches.map[ edge |
450                   edge.targetNode
451                 ].join("", "Or", "",
452                   [firstNodeOfBranch|firstNodeOfBranch.name]
453                 )
454             thenTask.owningThread = thread
455             thenTask.operands.addAll(
456               nonDefaultBranches.map [ edge |
457                 val proxyTask = threadedFactory.createProxyTask
458                 proxyTask.name = "Proxy" +
459                   (result.values.filter[ activityNode |
460                      activityNode == edge.targetNode
461                    ].toList.size + 1)
462                    + "For" + edge.targetNode.name
463                 proxyTask.owningThread = thread
464                 result.put(proxyTask, edge.targetNode)
465                 tasksNotNecessarilyExecuted.add(proxyTask)
466                 proxyTask
467               ])
468             conditionalTask.thenTask = thenTask
469             tasksNotNecessarilyExecuted.add(thenTask)
470           }
471         ]
472
473       }
474       default: {
475         // Default case is to transform the Node into an
    ExecutionTask.
476         val task = threadedFactory.createExecutionTask
477         task.name = "Execute_" + node.name
478         task.owningThread = thread
479
480         // Create the corresponding ModelProjection
481         val modelProjection =
482           projectionsFactory.createModelProjection
483         modelProjection.name =
```

```
484             projectionForExecution.name + "_" + node.name
485           modelProjection.moccElement = task
486           modelProjection.languageElement = node
487           modelProjections.modelProjections.add(modelProjection)
488         }
489       }
490
491       return result
492     }
493
494     private def Set<Set<ActivityEdge>> computePossiblePermutations(
495                                       List<ActivityEdge> edges) {
496       var Set<Set<ActivityEdge>> result = Sets.powerSet(edges.toSet)
497
498       // Check if one of the edges has the default guard
499       if (edges.exists[edge|edgeHasDefaultGuard(edge)]) {
500         // In that case, remove all the sets where the default edge
      is not present
501         val defaultEdge =
502           edges.findFirst[edge|edgeHasDefaultGuard(edge)]
503
504         result = result.filter [ set |
505           set.contains(defaultEdge)
506         ].toSet
507       }
508
509       return result
510     }
511
512     private def boolean edgeHasDefaultGuard(ActivityEdge edge) {
513       edge.guard instanceof LiteralString
514         && (edge.guard as LiteralString).value.equals("else")
515     }
516
517     private def boolean nodeIsAncestorOf(
518       ActivityNode candidateAncestor,
519       ActivityNode candidateSuccessor) {
520       if (candidateAncestor == candidateSuccessor) {
521         // If the two nodes are the same, the coherent result is
      false.
522         return false
523       } else  // If an outgoing edge leads to the candidate successor
524       if (candidateAncestor.outgoingEdges.exists [ edge |
525         edge.targetNode == candidateSuccessor
```

```
526       ]) {
527         return true
528       } else // If the candidate successor is a pin of the candidate
        ancestor
529       if (candidateAncestor instanceof Action &&
530         (candidateAncestor as Action).outputs.exists[ pin |
531           pin == candidateSuccessor
532         ]) {
533         return true
534       } else { // Recursivity with the correct navigation
535         val nextNodesToExplore = new ArrayList()
536         nextNodesToExplore.addAll(
537           candidateAncestor.outgoingEdges.map[edge|edge.targetNode]
538         )
539         if (candidateAncestor instanceof Action) {
540           nextNodesToExplore.addAll(
541             (candidateAncestor as Action).outputs
542           )
543         }
544         return nextNodesToExplore.exists[nextNode |
545           nodeIsAncestorOf(nextNode, candidateSuccessor)
546         ]
547       }
548     }
549
550     private def Map<ActivityNode, Pair<ForkNode, ActivityEdge>>
        findBranchesHoldingNodes(Activity activity,
551       Collection<Pair<ForkNode, JoinNode>> pairsOfForkAndJoin) {
552       val result = new HashMap()
553       val nodesToConsider = new ArrayList(activity.nodes)
554       nodesToConsider.addAll(
555         activity.nodes
556           .filter[node|node instanceof Action]
557           .map[node|(node as Action).outputs]
558           .flatten
559       )
560
561       nodesToConsider.forEach [ node |
562         val branch = findBranchFor(node, pairsOfForkAndJoin)
563         if (branch != null) {
564           result.put(node, branch)
565         } else {
566           // It's on the main thread
567         }
```

```
568     ]
569
570     return result
571   }
572
573   private def Pair<ForkNode, ActivityEdge> findBranchFor(
       ActivityNode node,
574     Collection<Pair<ForkNode, JoinNode>> pairsOfForkAndJoin) {
575     switch node {
576       JoinNode case true:
577       // A JoinNode is on the same branch as its corresponding
      ForkNode
578       {
579         return findBranchFor(pairsOfForkAndJoin
580           .findFirst[ pair |
581             pair.value == node
582           ].key,
583           pairsOfForkAndJoin)
584       }
585       OutputPin case true:
586       // A Pin is on the same branch as its owning node
587       {
588         return findBranchFor(
589           node.eContainer as Action, pairsOfForkAndJoin
590         )
591       }
592       ActivityNode case node.incomingEdges.exists[ edge |
593         edge.sourceNode instanceof ForkNode
594       ]:
595       // The node is the beginning of a branch
596       {
597         return new Pair(node.incomingEdges.findFirst[ edge |
598           edge.sourceNode instanceof ForkNode
599           ].sourceNode,
600           node.incomingEdges.findFirst[ edge |
601             edge.sourceNode instanceof ForkNode
602           ])
603       }
604       default:
605       // In all other cases, return one of the incoming edges'
      source
606       {
607         if (node.incomingEdges.isEmpty) {
608           return null
```

```
609         } else {
610           return findBranchFor(
611             node.incomingEdges.head.sourceNode, pairsOfForkAndJoin
612           )
613         }
614       }
615     }
616
617   }
618
619   // Go through an Activity to find out its Fork/Join couples.
620   private def <T1 extends ActivityNode, T2 extends ActivityNode>
       Collection<Pair<T1, T2>> findNodePairs(Activity activity,
       Class<T1> t1, Class<T2> t2) {
621     val result = new ArrayList<Pair<T1, T2>>()
622
623     val forkNodes = activity.nodes
624       .filter[node|t1.isAssignableFrom(node.class)]
625       .map[node|node as T1]
626       .toList
627     val joinNodes = activity.nodes
628       .filter[node|t2.isAssignableFrom(node.class)]
629       .map[node|node as T2]
630       .toList
631
632     if (forkNodes.size == 1 && joinNodes.size == 1) {
633       result.add(new Pair(forkNodes.get(0), joinNodes.get(0)))
634
635     // Enough for the example Activity
636     } else {
637
638       val remainingForkNodes = new ArrayList(forkNodes)
639       val remainingJoinNodes = new ArrayList(joinNodes)
640
641       // For the outer-most ForkNode, there is at most forkNodes.
      size-1 ForkNode/JoinNode couples before its own JoinNode.
642       for (depth : 0 ..< forkNodes.size) {
643
644         // Find the ForkNodes for which there is depth other
      ForkNodes and depth JoinNodes on its path to a JoinNode.
645         val Collection<Pair<T1, T2>> currentInnerMostCouples =
646           findNodeCouplesBasedOnDepth(
647             remainingForkNodes, remainingJoinNodes, depth
648           )
```

```
649
650         // The first JoinNode encountered is the one we are
    looking for
651         // Add this couple to the result and remove them from the
     pool of remaining forks and joins.
652         result.addAll(currentInnerMostCouples)
653         remainingForkNodes.removeAll(
654           currentInnerMostCouples
655             .map[pair|pair.key]
656         )
657         remainingJoinNodes.removeAll(
658           currentInnerMostCouples
659             .map[pair|pair.value]
660         )
661       }
662     }
663     return result
664   }
665
666   // Returns the ForkNodes for which there is $depth other
     forknodes on their path to a JoinNode. Also associates the
     JoinNode in question.
667   private def <T1 extends ActivityNode, T2 extends ActivityNode>
     Collection<Pair<T1, T2>> findNodeCouplesBasedOnDepth(
     Collection<T1> forkNodes,
668     Collection<T2> joinNodes, int depth) {
669     val result = new ArrayList()
670
671     forkNodes.forEach [ forkNode |
672       var stop = false
673       var numberOfCouplesEncountered = 0
674       val Queue<ActivityEdge> pathsToExplore = new LinkedList(
    forkNode.outgoingEdges)
675       // Keep going until
676       // Either the number of couples encountered is > depth
677       // Or there is no more paths to explore
678       while (!stop) {
679         // To be completed
680         // Navigate the paths until we encounter a JoinNode and
    $depth ForkNode and JoinNode have been met.
681         stop =
682           numberOfCouplesEncountered > depth || pathsToExplore.
    isEmpty
683       }
```

```
684     ]
685
686     return result
687   }
688
689 }
```

E.4 Projections

The Projections, as presented in Chapter 4, are specified using the Projections metalanguage (whose textual concrete syntax is available in Appendix H). They are shown on Listing 4.6.

E.5 Communication Protocol

The Communication Protocol, specified using GEL and exploiting the Projections defined in the previous section, are shown on Listing 4.7.

F

Execution of the Example fUML Model Using the Threading MoC

In Chapter 4 we have presented how to use a previously-defined concurrency-aware xDSML as the MoC of an xDSML. In this appendix, we detail the realization of the execution of the example fUML Activity used throughout Chapters 3 and 4. The fUML Activity is shown on Figure 4.1.

The execution is realized using a definition of fUML whose MoC is not Event Structures, but the thread-based language presented in Subsection 4.2.2 of Chapter 4. This implementation of fUML is shown in Appendix E.

For the Threading xDSML, the graphical syntax is as follows. Each `Thread` is represented as a node containing its `Instructions`. The main thread is designated by a "*" appended to its name. When an Instruction consists in starting a thread or in waiting for a thread, there is an arrow from the instruction to the thread, respectively from the thread to the instruction. The animation layer is captured in the background colour of the instructions and threads. A grey thread is inactive, while a green thread is active, and an orange thread is a thread that has finished all its instructions. An orange instruction has been executed, a green one may be executed (possibly conditionally) and a grey one may not be.

Figure F.1 shows the MoCApplication for our example fUML Activity. This MoCApplication is a model conforming to the Threading xDSML we have defined, and is obtained automatically thanks to $\mathcal{T}_{\text{fUML}\rightarrow\text{Threading}}$. A textual concrete syntax for this model is shown on Listing 4.1 of Chapter 4.

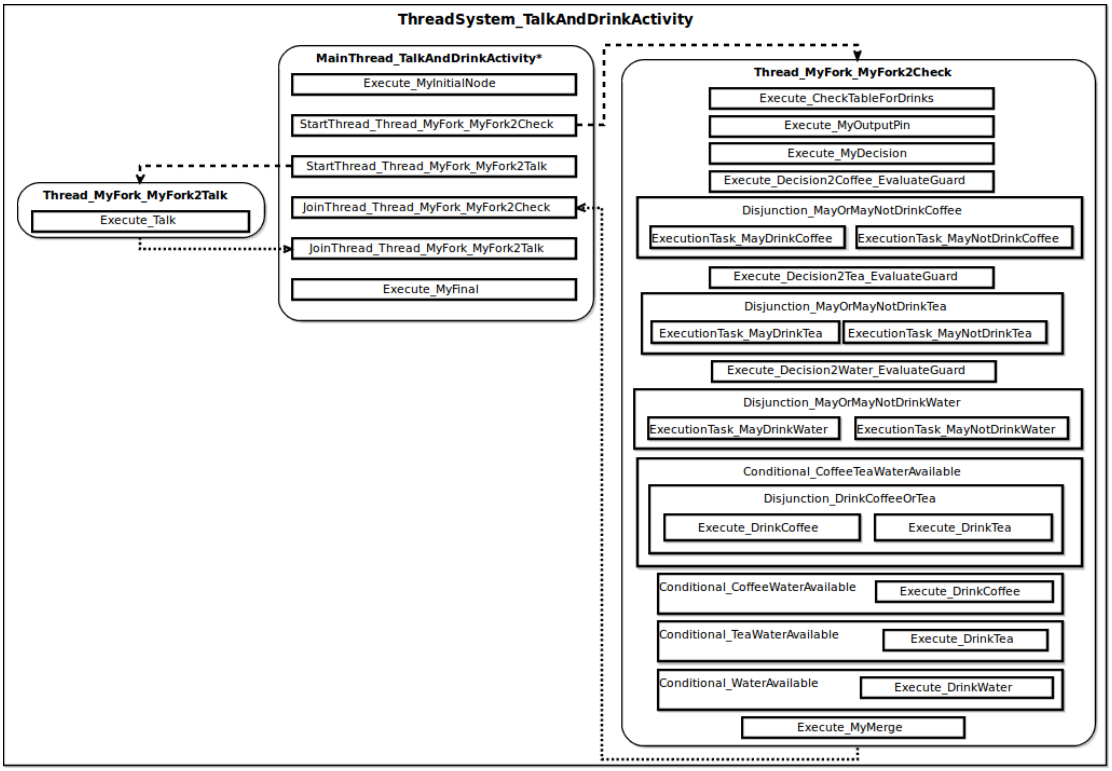

Figure F.1: MoCApplication of the example fUML Activity, based on the Threading MoC.

The initial view of the Modeling Workbench when launching the execution of the example fUML Activity is shown on Figure F.2. There are two active Execution Engine: one for fUML, and one for the Threading xDSML. The second one is used as the Solver of the first engine, as explained in Chapter 4. We focus on the execution of the fUML Activity, and therefore we do not show the occurring MappingApplications of the Threading model or its underlying Event Structure.

The first possible execution step does not have occurrences of MappingApplications for fUML, but it does have an occurrence of the MappingApplication corresponding to starting the main Thread. Figure F.3 shows the result of executing this step.

Now that the main Thread has been activated, its first instruction "Execute_MyInitialNode" can be executed. This is matched by the Communication Protocol of fUML and mapped

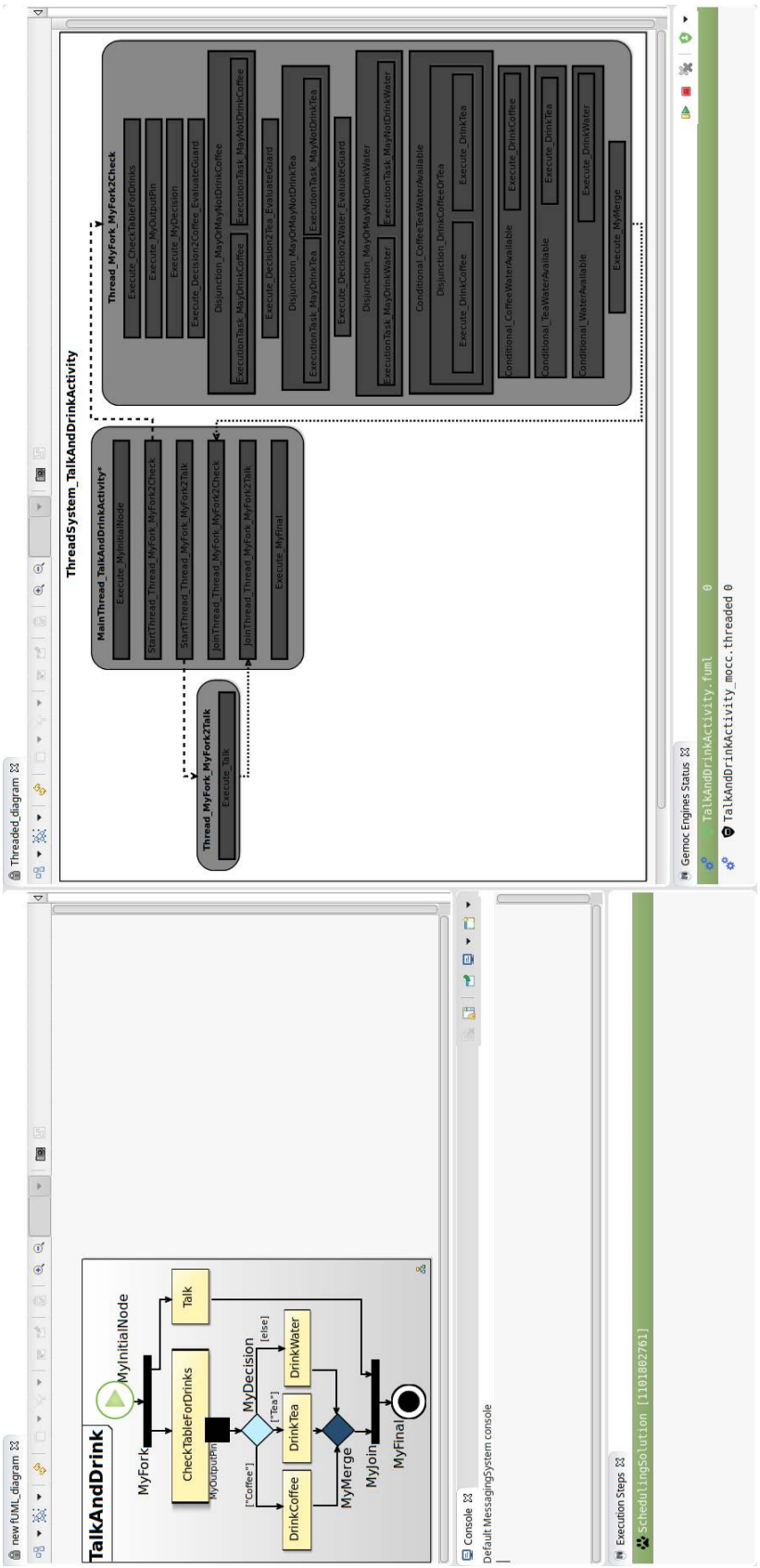

Figure F.2: Step 0 – Initial view of the Modeling Workbench when launching the execution of the example fUML Activity using the Threading Model of Concurrency.

to the fUML Execution Function call of “MyInitialNode.execute()”. Figure F.4 shows the result of executing this step.

The first instruction of the main thread has been executed, and so has the InitialNode in the fUML model. In the next possible execution step, executing the second instruction

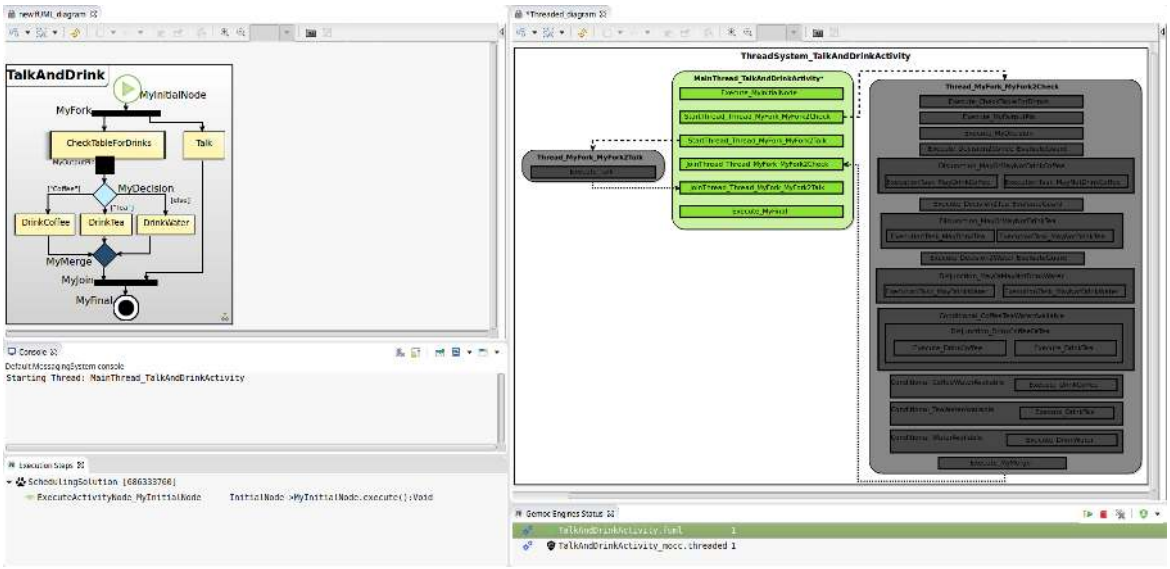

Figure F.3: Step 1 – Execution of the example fUML Activity using the Threading Model of Concurrency.

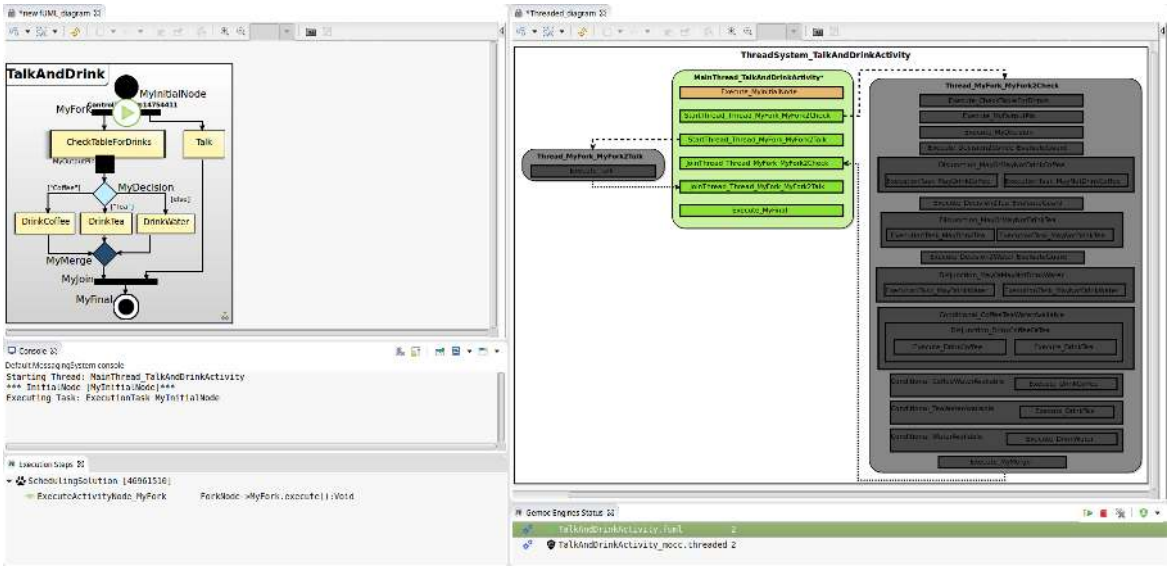

Figure F.4: Step 2 – Execution of the example fUML Activity using the Threading Model of Concurrency.

"StartThread_Thread_MyFork_MyFork2Check" is mapped to the Execution Function call corresponding to "MyFork.execute()". Figure F.5 shows the result of executing this step.

The Thread corresponding to the drinking part of the activity has been activated. There are now two active Threads with instructions left to execute, therefore we have multiple possible execution steps. We select the step that does both at the same time, *i.e.*, executing the next instruction of the main Thread (which will activate the last Thread) and executing the first instruction of the second Thread ("Execute_CheckTableForDrinks"). Figure F.6 shows the result of executing this step.

Now that the three threads are active, there are even more possible execution steps. Once again, we select the step with the maximum activity. It corresponds to the execution of three instructions ("JoinThread_MyFork_MyFork2Check" in the main Thread, "Execute_MyOutputPin" in the second Thread, and "Execute_Talk" in the third Thread). Two of them are mapped by the Communication Protocol of fUML to Execution Function calls ("MyOutputPin.execute()" and "Talk.execute()"). Figure F.7 shows the result of executing this step.

The main Thread is now blocked, waiting for the second thread to finish. The third Thread has finished its instructions so among the possible execution steps is the end of that thread. Once again we select the execution step with the most activity, which means that the third thread will terminate and the next instruction of the second thread will be executed. Figure F.8 shows the result of executing this step.

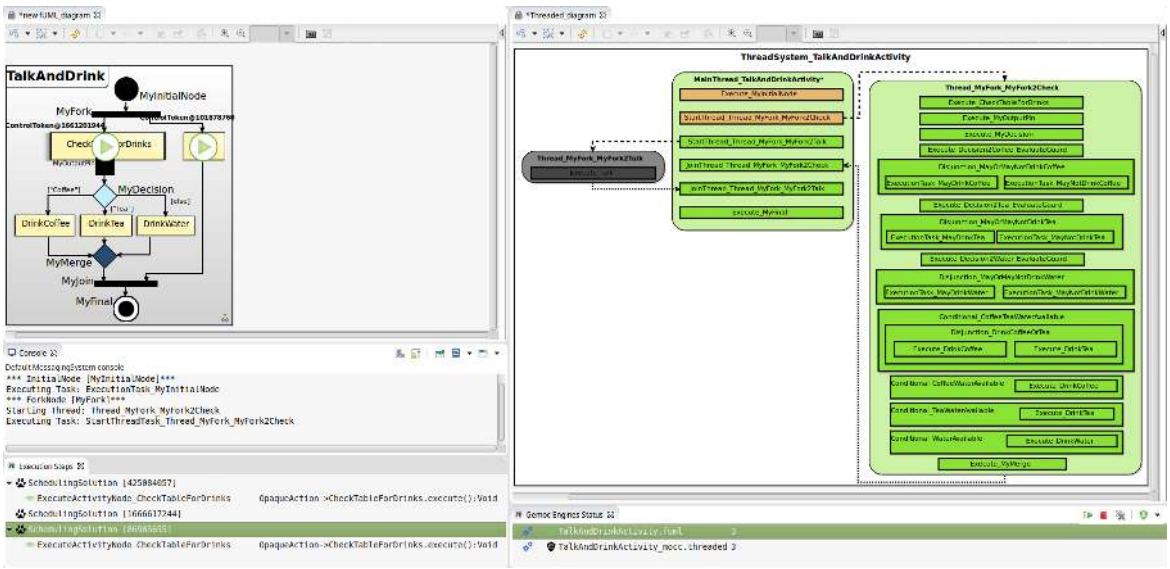

Figure F.5: Step 3 – Execution of the example fUML Activity using the Threading Model of Concurrency.

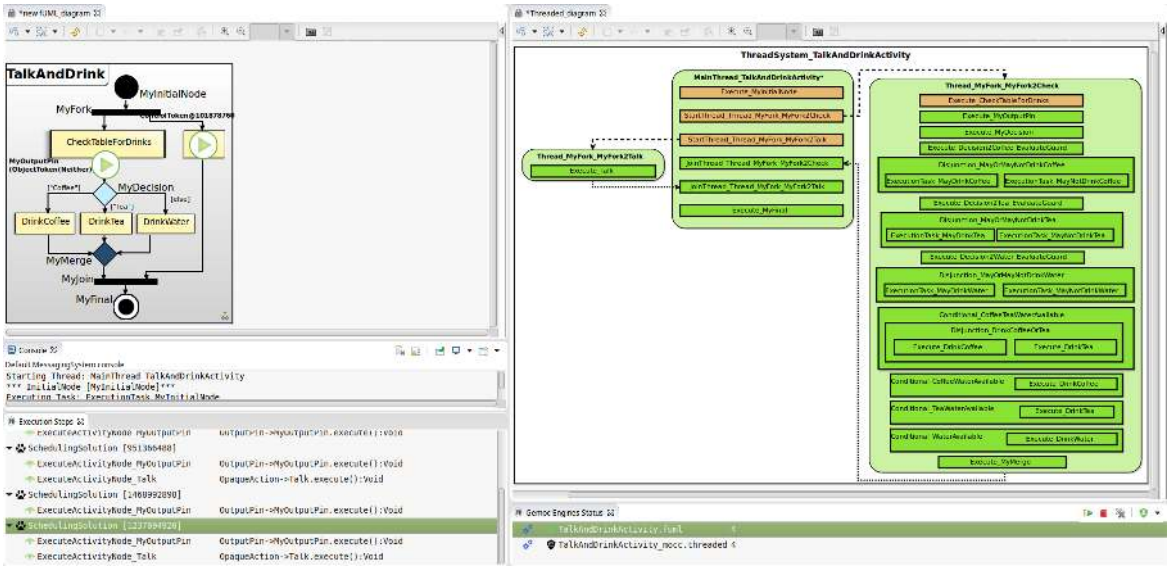

Figure F.6: Step 4 – Execution of the example fUML Activity using the Threading Model of Concurrency.

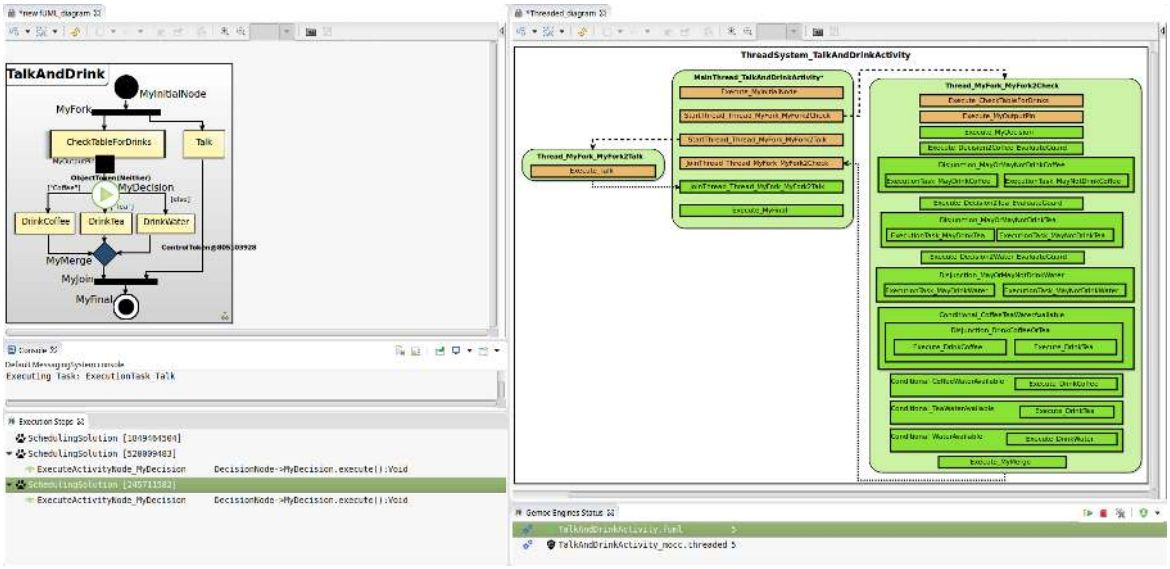

Figure F.7: Step 5 – Execution of the example fUML Activity using the Threading Model of Concurrency.

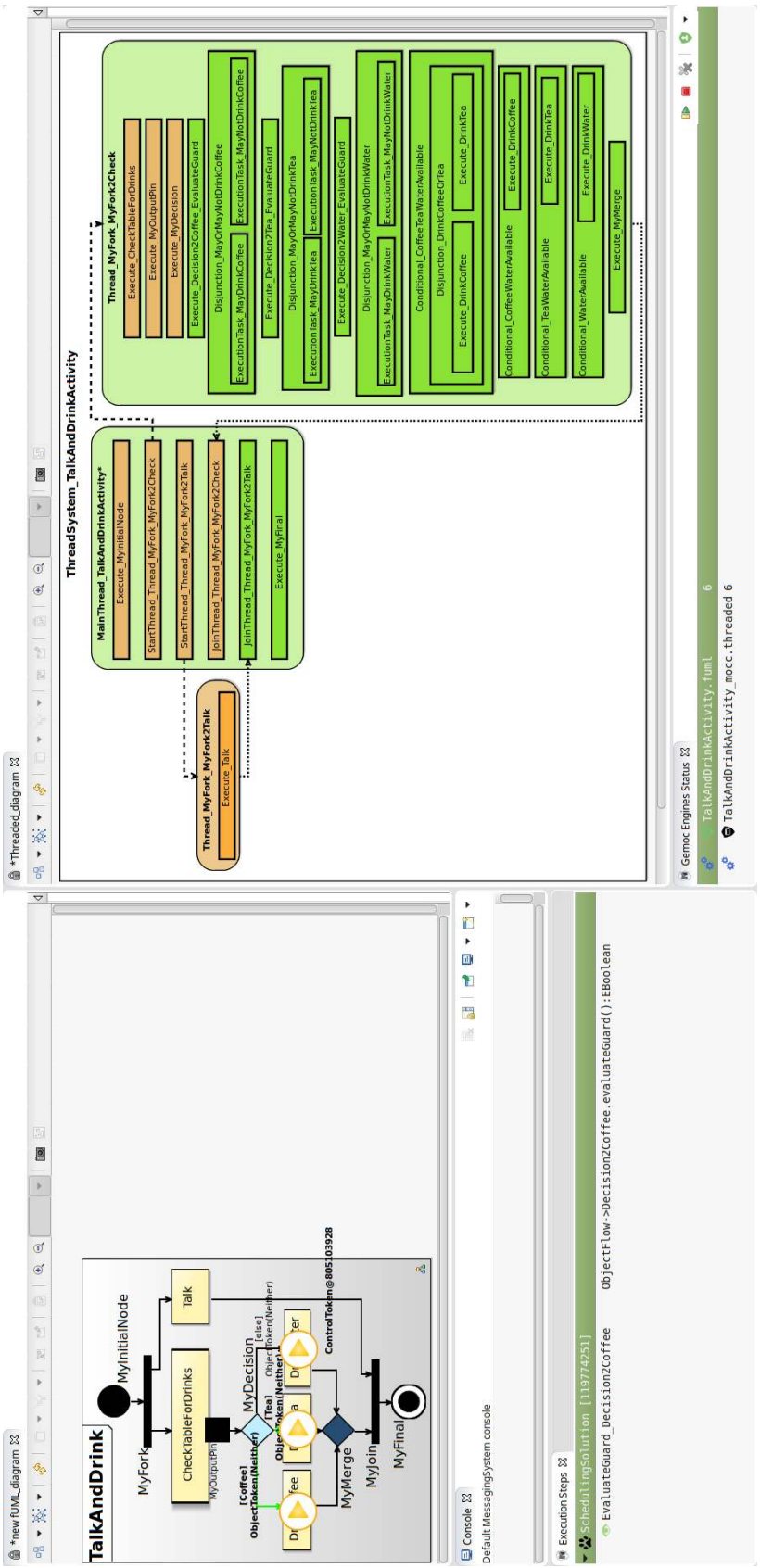

Figure F.8: Step 6 – Execution of the example fUML Activity using the Threading Model of Concurrency.

Next, we keep on going in the drinking part of the activity. The DecisionNode is a bit more complex due to the presence of guards. Taking their results into account is one of the main features presented in Chapter 3 (see Section 3.6). This is represented, in the Threading language, as follows. One instruction corresponds to evaluating the guard. Then, a disjunction between two instructions corresponds to the consequence of the result of the guard evaluation. Later on, these will be used to determine whether or not the corresponding branch may be executed.

The next 6 steps correspond to the evaluation of each guard (2 steps per guard). See Figures F.9, F.10, F.11, F.12, F.13, F.14.

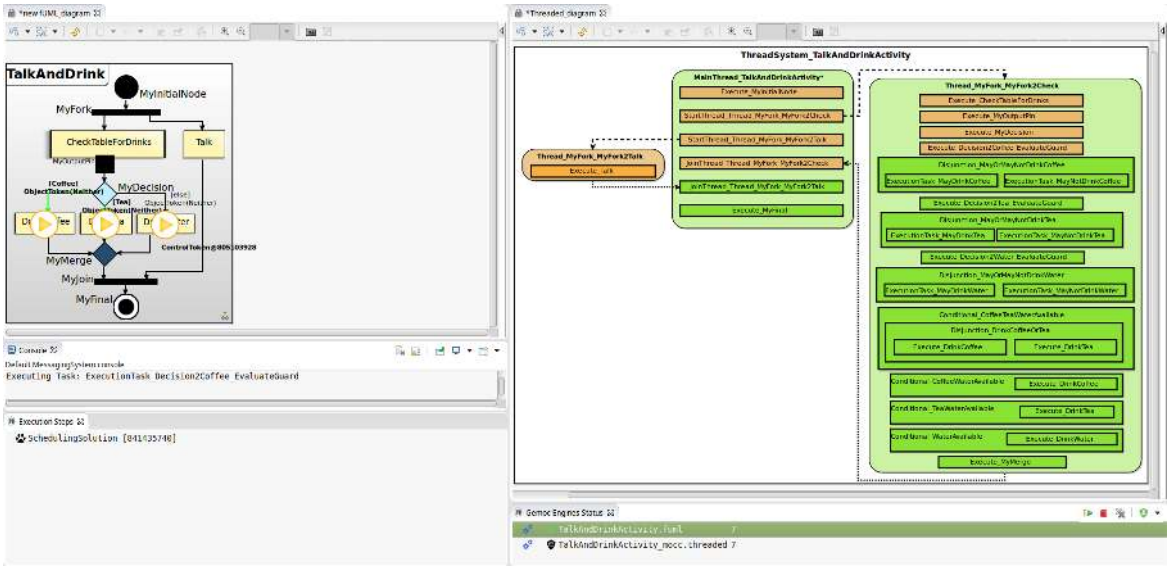

Figure F.9: Step 7 – Execution of the example fUML Activity using the Threading Model of Concurrency.

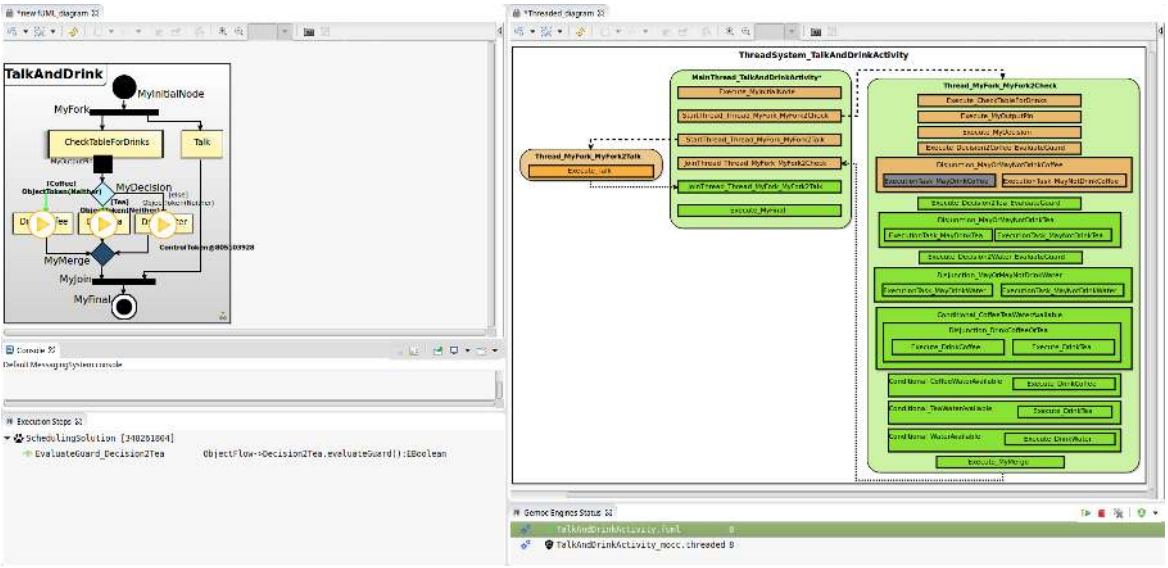

Figure F.10: Step 8 – Execution of the example fUML Activity using the Threading Model of Concurrency.

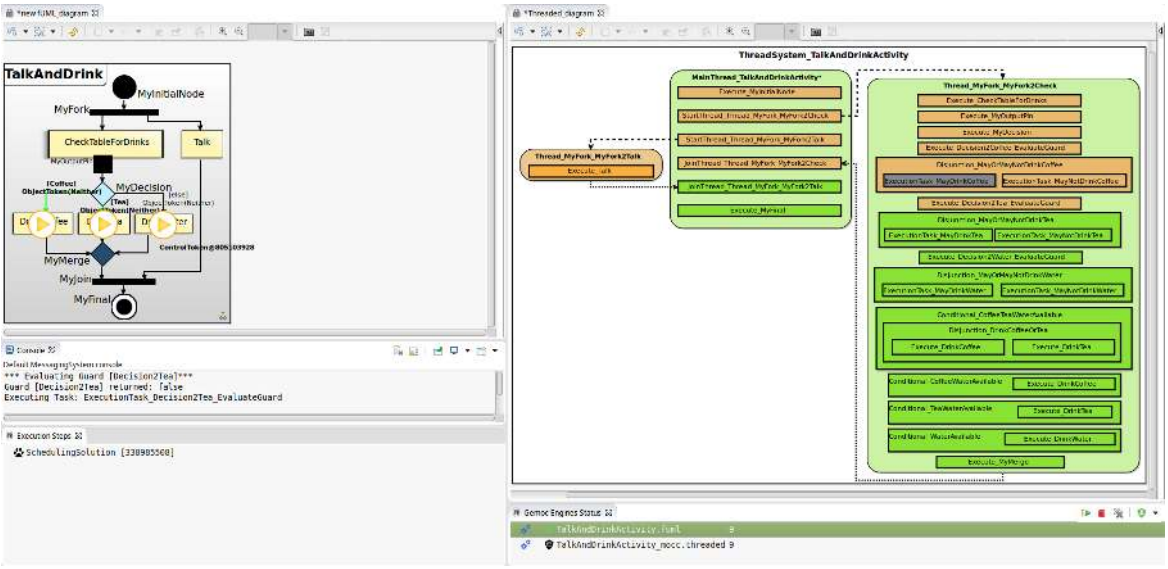

Figure F.11: Step 9 – Execution of the example fUML Activity using the Threading Model of Concurrency.

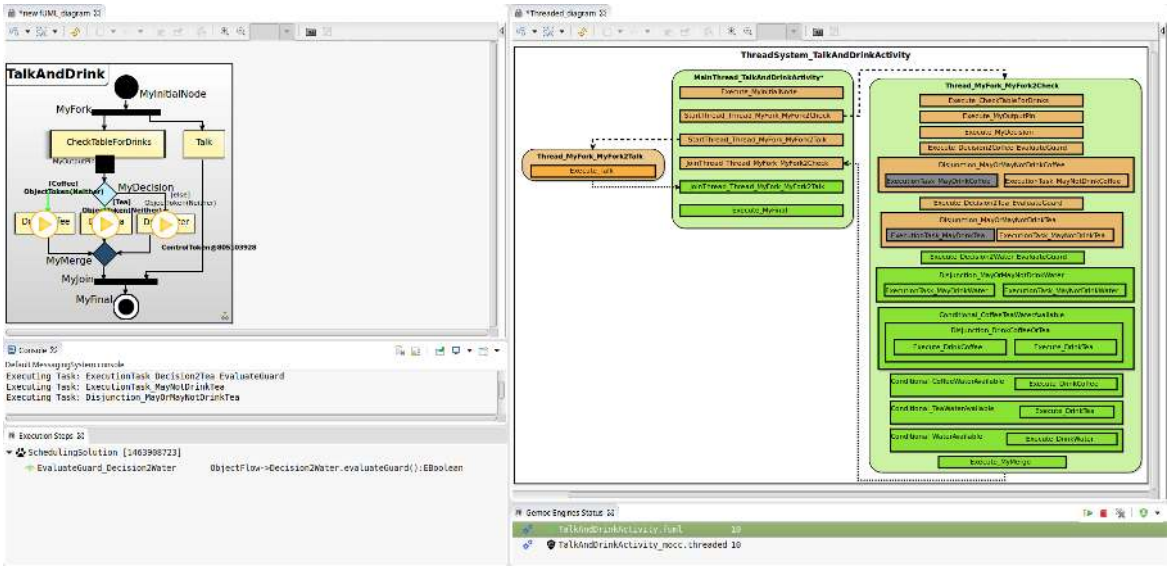

Figure F.12: Step 10 – Execution of the example fUML Activity using the Threading Model of Concurrency.

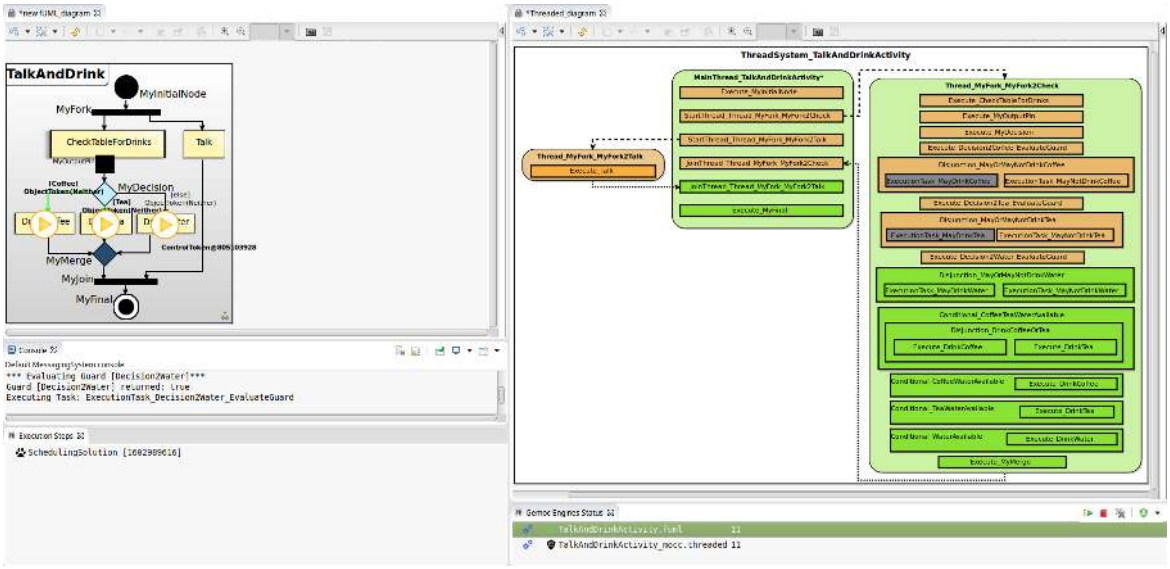

Figure F.13: Step 11 – Execution of the example fUML Activity using the Threading Model of Concurrency.

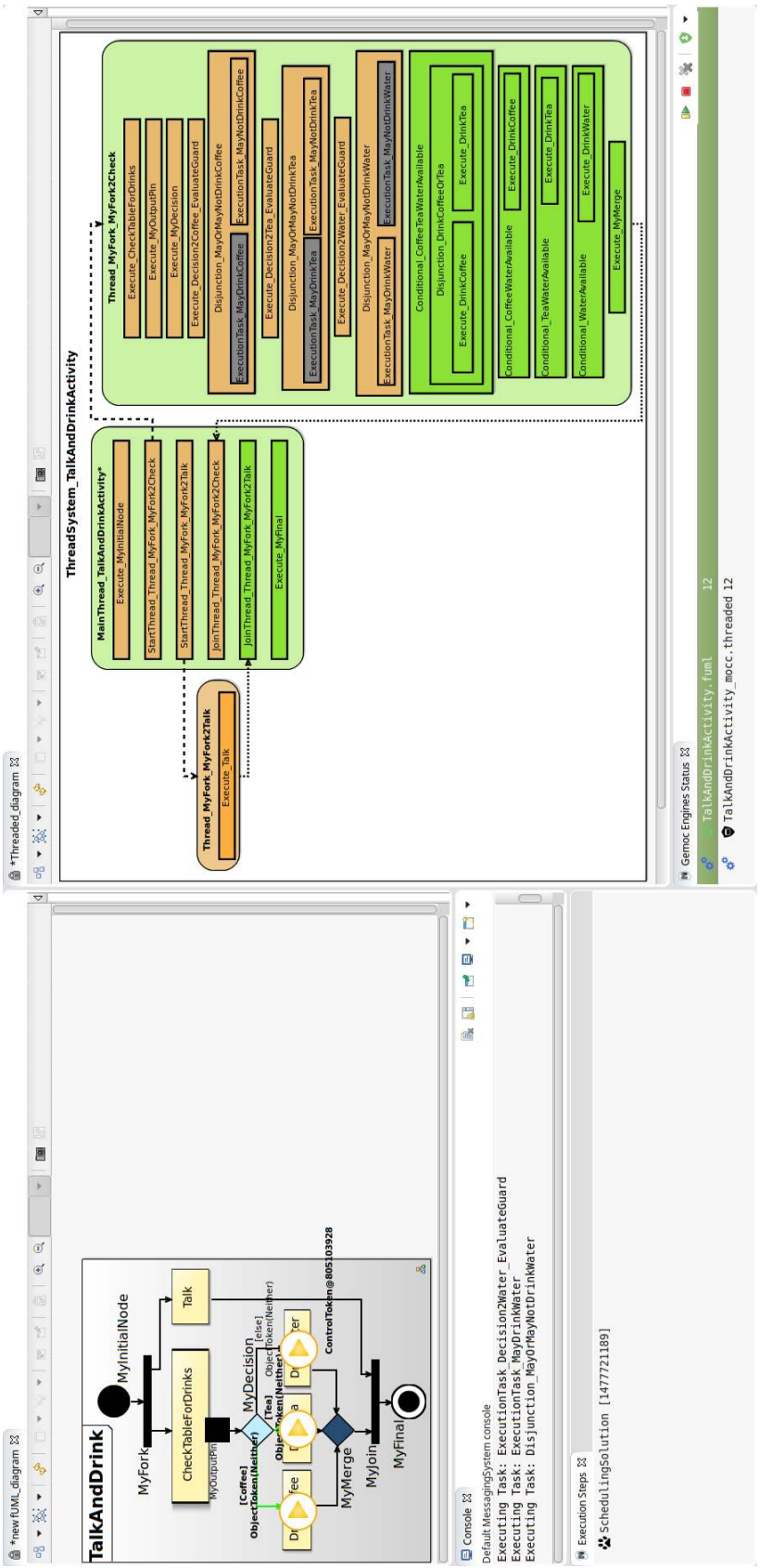

Figure F.14: Step 12 – Execution of the example fUML Activity using the Threading Model of Concurrency.

In the next 3 steps, the instructions correspond to the conditionals used to check which drink will be drunk. In our execution scenario, we have found no coffee or tea on the table so we will ultimately drink water. See Figures F.15 and F.16 for the conditionals that fail. Figure F.17 shows the conditional that will lead us to drinking water.

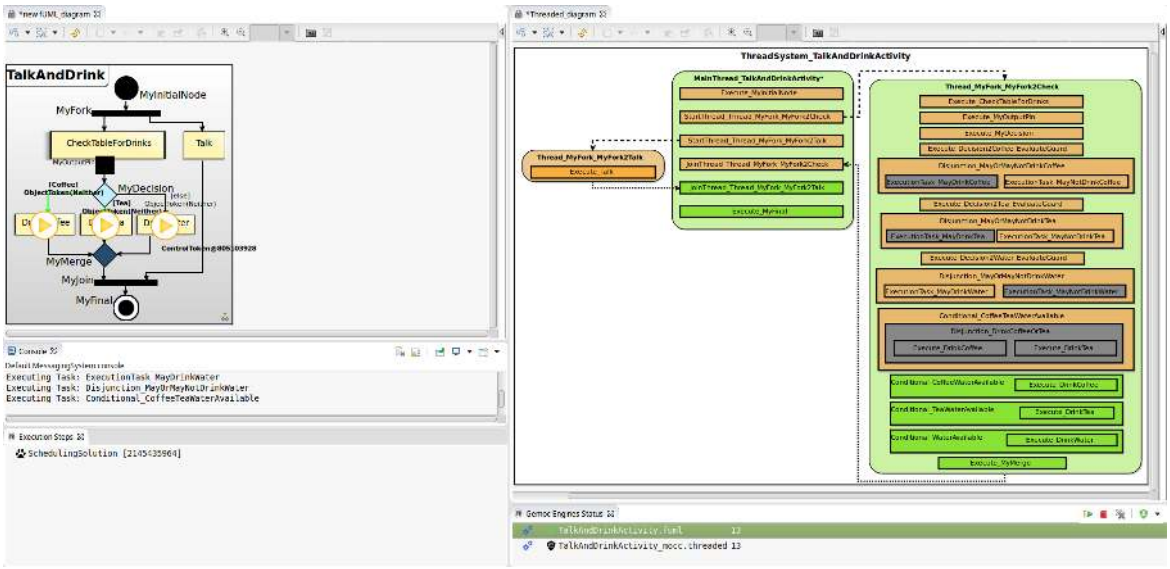

Figure F.15: Step 13 – Execution of the example fUML Activity using the Threading Model of Concurrency.

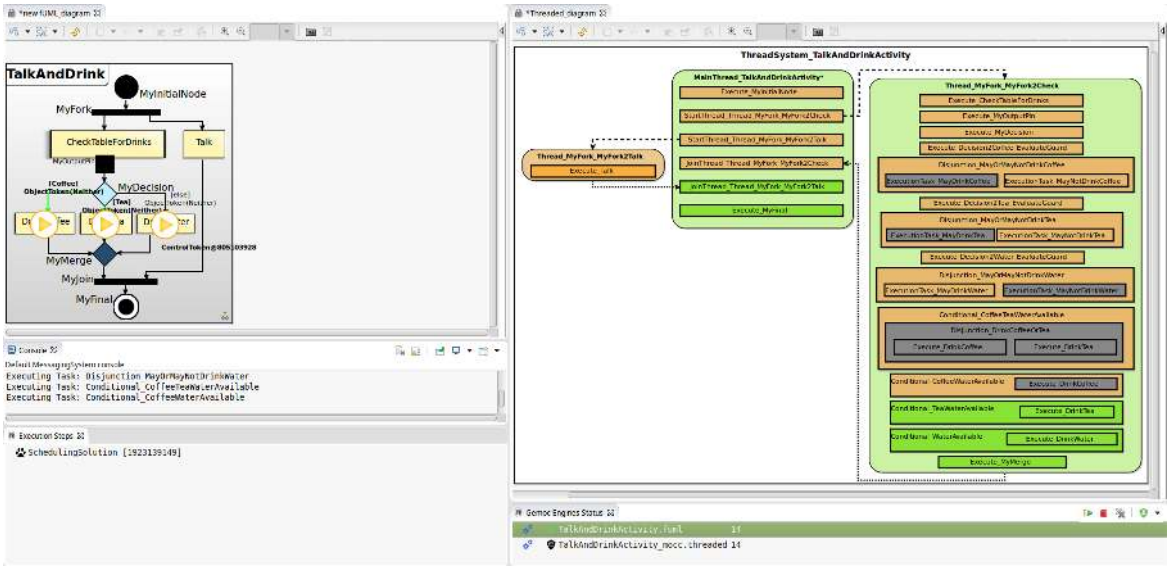

Figure F.16: Step 14 – Execution of the example fUML Activity using the Threading Model of Concurrency.

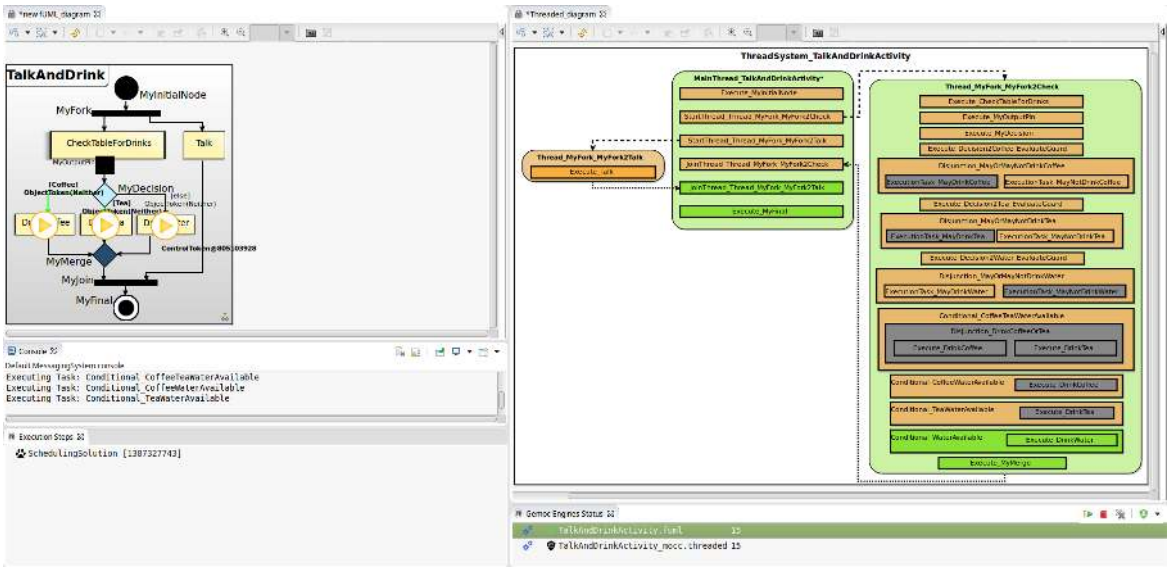

Figure F.17: Step 15 – Execution of the example fUML Activity using the Threading Model of Concurrency.

Figure F.18 shows that the instruction (allowed since the conditional's condition was validated) "Execute_DrinkWater" is mapped by the Communication Protocol of fUML to the Execution Function call "DrinkWater.execute()".

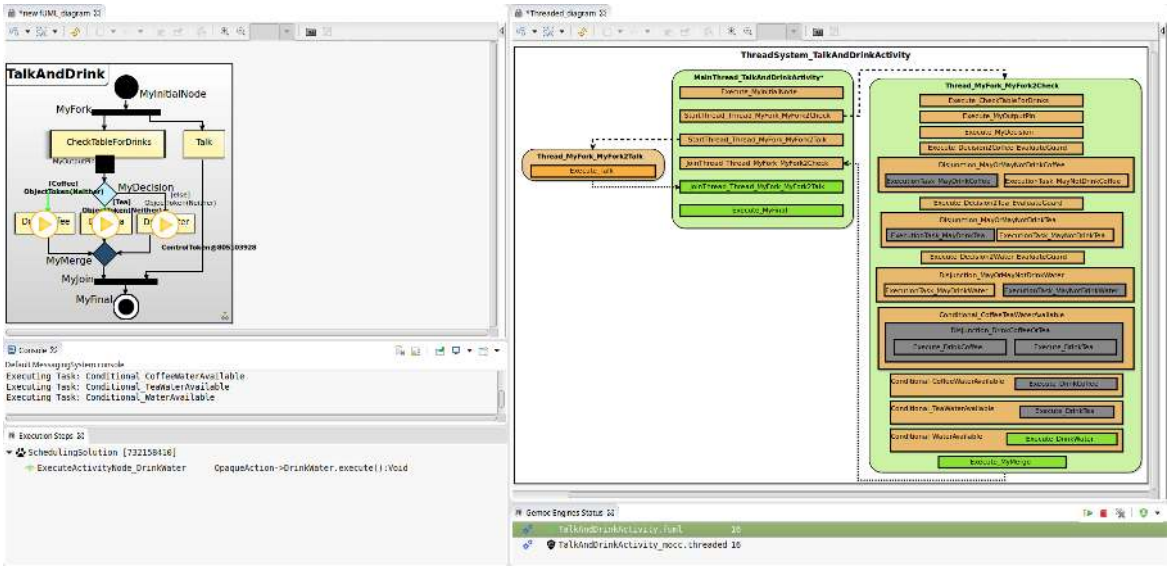

Figure F.18: Step 16 – Execution of the example fUML Activity using the Threading Model of Concurrency.

Figure F.19 shows that once one of the branches of the DecisionNode has been executed, the MergeNode may be executed.

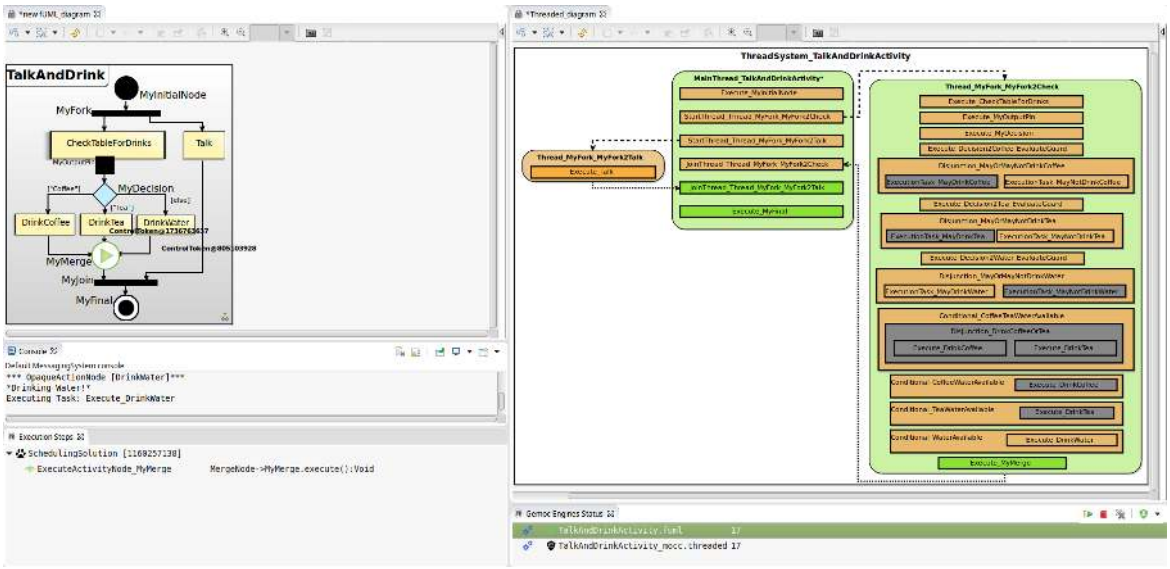

Figure F.19: Step 17 – Execution of the example fUML Activity using the Threading Model of Concurrency.

In Figure F.20, all the instructions of the second Thread have been executed, therefore the Thread will be terminated.

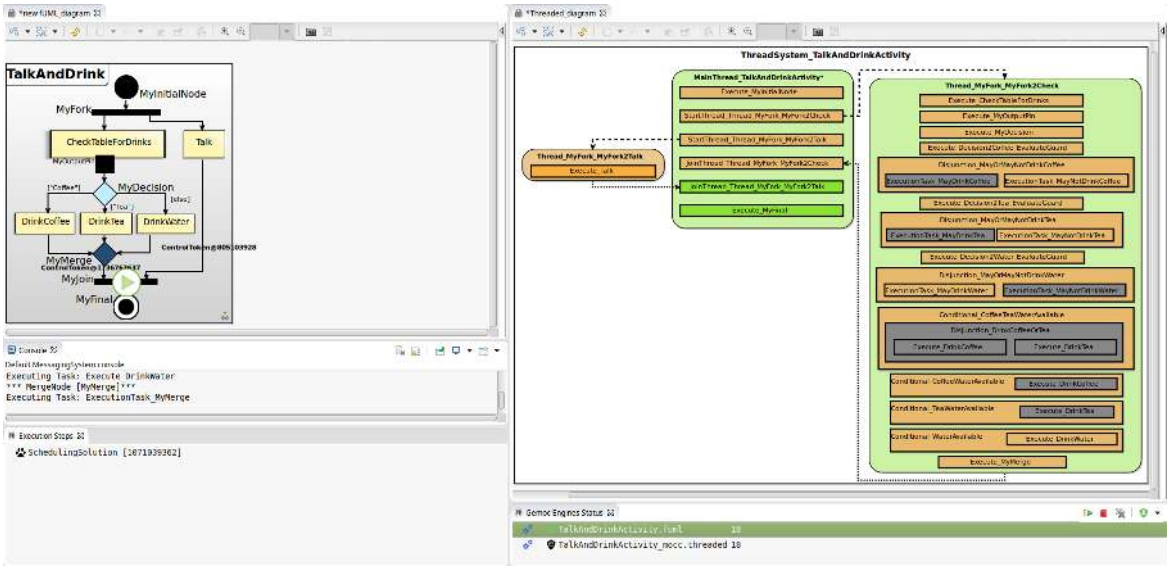

Figure F.20: Step 18 – Execution of the example fUML Activity using the Threading Model of Concurrency.

In Figure F.21, the second Thread has been terminated. This unblocks the main Thread, which may now block on the third Thread having been executed. This corresponds to the execution of the JoinNode in the fUML model.

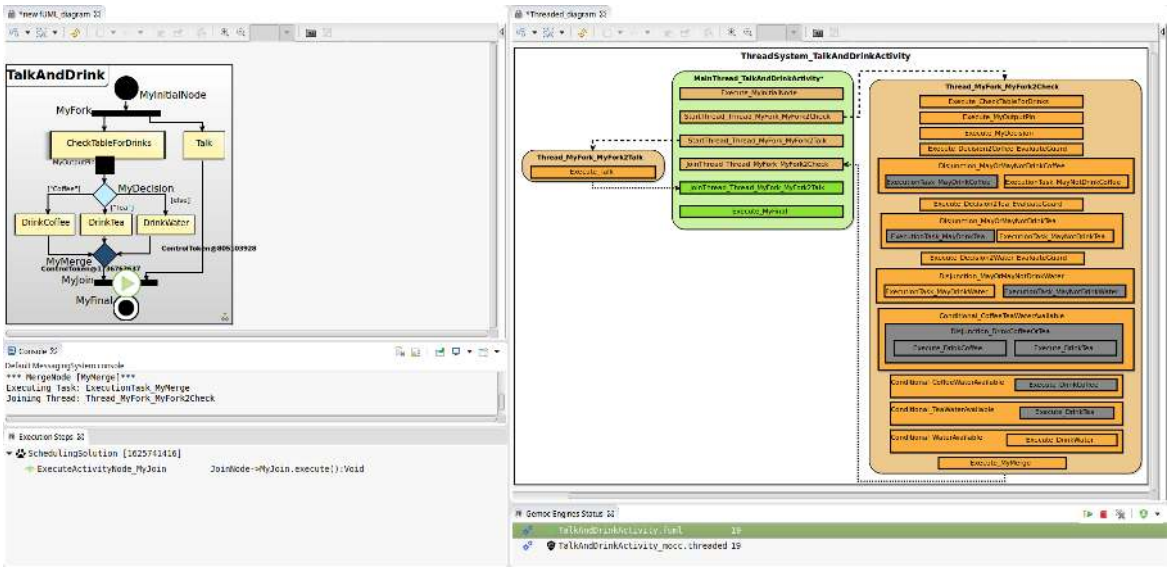

Figure F.21: Step 19 – Execution of the example fUML Activity using the Threading Model of Concurrency.

Figure F.22 shows that the final instruction may be executed, corresponding to the execution of the FinalNode in the fUML model.

Finally in Figure F.23, the main thread may be terminated now that all its instructions have been executed. The final state is shown on Figure F.24.

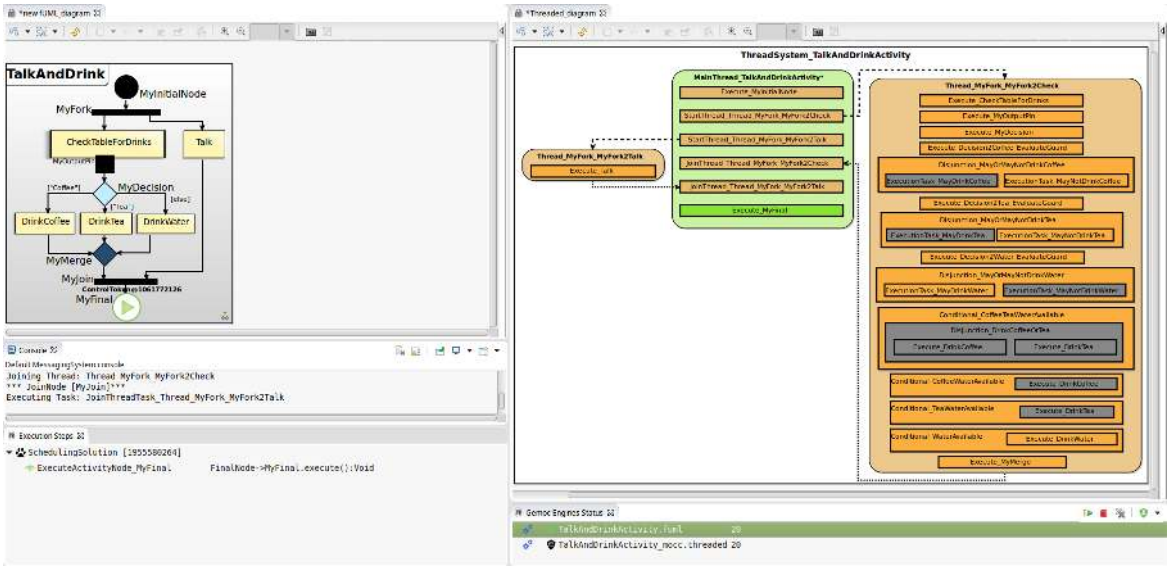

Figure F.22: Step 20 – Execution of the example fUML Activity using the Threading Model of Concurrency.

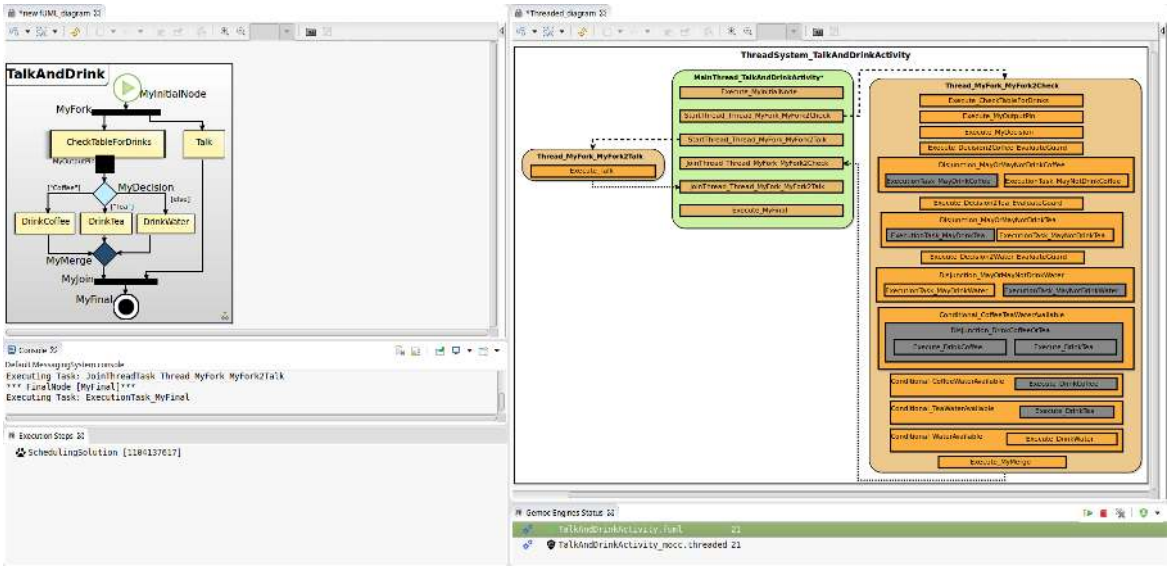

Figure F.23: Step 21 – Execution of the example fUML Activity using the Threading Model of Concurrency.

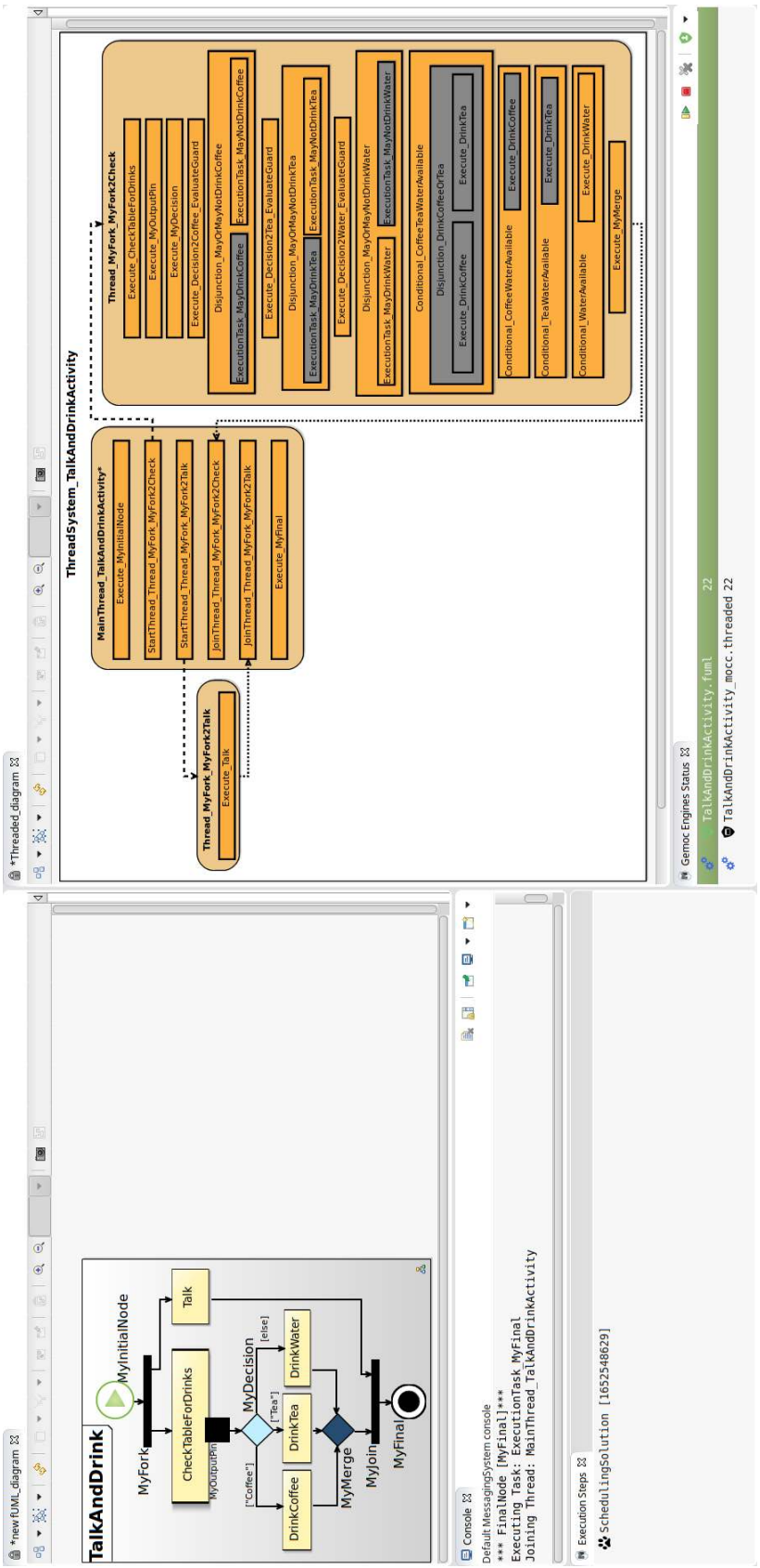

Figure F.24: Step 22 – Execution of the example fUML Activity using the Threading Model of Concurrency.

Textual Concrete Syntax of the GEMOC Events Language (GEL)

There are two approaches to using Xtext [7]. The first one consists in specifying the concrete syntax, and let Xtext generate the corresponding abstract syntax as an Ecore metamodel. Sometimes, getting the "right" abstract syntax with the strategy can be difficult or frustrating. The second one consists in first designing the abstract syntax of the language, and then designing the Xtext concrete syntax over it. This ensures that the metamodel (which is exploited by other facilities through its APIs) respects a particular structure and naming convention. For GEL, we used the latter, as the textual concrete syntax was added only after first proofs of concepts of the integration of GEL into the GEMOC Studio were successful. The Abstract Syntax used can be found on Figure 3.44.

The following listing is the full Xtext textual concrete syntax of GEL. We reuse another Xtext-based language called GExpressions which provides the means to use basic arithmetic and navigation expressions on models and metamodels with an OCL-like syntax. It is used by several metalanguages developed during the ANR INS GEMOC project.

Listing G.1: The Xtext textual concrete syntax of GEL.

```
grammar org.gemoc.gel.GEL with org.gemoc.gel.gexpressions.xtext.GExpressions

// Used for the Abstract Syntax (and Semantic Rules) of the xDSML
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
// Abstract Syntax of GEL
import "http://www.gemoc.org/gel/GEL" as gel
// Used for the MoCMapping
import "http://fr.inria.aoste.timesquare.ecl" as ecl
// Used for the xDSML as MoC feature (cf. Chapter 4)
import "http://www.gemoc.org/gel/projections" as projections

//----------------------
// Main elements
//----------------------
DomainSpecificEventsSpecification returns gel::DomainSpecificEventsSpecification:
 {gel::DomainSpecificEventsSpecification}
 (imports += ImportStatement)*
 events+=(DomainSpecificEvent)*
;

ImportStatement returns gel::ImportStatement:
 'import' importURI=STRING
;

DomainSpecificEvent returns gel::DomainSpecificEvent:
 AtomicDomainSpecificEvent | CompositeDomainSpecificEvent
;

AtomicDomainSpecificEvent returns gel::DomainSpecificEvent:
 {gel::AtomicDomainSpecificEvent}
 (visibility = Visibility)?
 'DSE' name=ID ':'
 'upon' uponMoccEvent=MoccEvent
 (executionKind=ExecutionKind executionFunction=ExecutionFunction
  ('feedback' ':' feedbackPolicy=FeedbackPolicy 'end')?
 )?
 ('raises' raisedMoccEvent=MoccEvent)?
 'end'
;
// End
```

```
41
42 //---------------------------------
43 // MoCMapping elements
44 //---------------------------------
45 MoccEvent returns gel::MoccEvent:
46  EclEvent | GelEvent
47 ;
48
49 // When using EventType Structures through ECL
50 EclEvent returns gel::MoccEvent:
51  {gel::EclEvent}
52  eventReference = [ecl::ECLDefCS|QualifiedName]
53 ;
54
55 // When using an xDSML as the MoC
56 GelEvent returns gel::GelEvent:
57  {gel::GelEvent}
58  'event' eventReference = [gel::DomainSpecificEvent|QualifiedName
      ] 'with' projection=[projections::LanguageProjection|ID]
59 ;
60 // End
61
62 //--------------------------------------
63 // Semantic Rules elements
64 //--------------------------------------
65 ExecutionFunction returns gel::ExecutionFunction:
66  Kermeta3ExecutionFunction
67 ;
68
69 Kermeta3ExecutionFunction returns gel::ExecutionFunction:
70  {gel::Kermeta3ExecutionFunction}
71  navigationPathToOperation = GExpression
72  (callKind=CallKind)?
73  ('returning'
74  result=ExecutionFunctionResult)?
75 ;
76
77 ExecutionFunctionResult returns gel::ExecutionFunctionResult:
78  {gel::ExecutionFunctionResult}
79  name=ID
80 ;
81 // End
82
83 //----------------------------------------
```

```
84 // Feedback Protocol elements
85 // (cf. Section 3.6)
86 //-----------------------------------------
87 FeedbackPolicy returns gel::FeedbackPolicy:
88  {gel::FeedbackPolicy}
89  (rules += FeedbackRule)*
90  defaultRule = DefaultFeedbackRule
91 ;
92
93 FeedbackRule returns gel::FeedbackRule:
94  {gel::FeedbackRule}
95  '[' filter=FeedbackFilter ']'
96  '=>' consequence=FeedbackConsequence
97 ;
98
99 DefaultFeedbackRule returns gel::FeedbackRule:
100  {gel::FeedbackRule}
101  'default' '=>' consequence=FeedbackConsequence
102 ;
103
104 FeedbackFilter returns gel::FeedbackFilter:
105  {gel::FeedbackFilter}
106   body=GExpression
107 ;
108
109 FeedbackConsequence returns gel::FeedbackConsequence:
110  {gel::FeedbackConsequence}
111  'allow' moccEvent=MoccEvent
112 ;
113 // End
114
115 //-------------------------------------------
116 // Composite Mapping elements
117 // (cf. Subsection 3.10.4)
118 //-------------------------------------------
119 CompositeDomainSpecificEvent returns gel::DomainSpecificEvent:
120  {gel::CompositeDomainSpecificEvent}
121  (visibility = Visibility)?
122  'Composite' name=ID ':'
123   (unfoldingStrategy = UnfoldingStrategy)?
124   body = DomainSpecificEventsPattern
125  'end'
126 ;
127
```

```
128 UnfoldingStrategy returns gel::UnfoldingStrategy:
129  {gel::UnfoldingStrategy}
130  'forall' '{'
131  (localVariables+=LocalVariable)+
132  '}' 'where' '{'
133  (instantiationPredicates+=InstantiationPredicate)+
134  '}'
135 ;
136
137 LocalVariable returns gel::LocalVariable:
138  {gel::LocalVariable}
139  name=ID ':' type=[ecore::EClassifier|QualifiedName]
140 ;
141
142 InstantiationPredicate returns gel::InstantiationPredicate:
143  {gel::InstantiationPredicate}
144  body=GExpression
145 ;
146
147 DomainSpecificEventsPattern returns gel::
       DomainSpecificEventsPattern:
148  LogicalSequence
149 ;
150
151 LogicalSequence returns gel::DomainSpecificEventsPattern:
152  CoincidencePattern ({gel::LogicalSequence.leftOperand=current}=> '
       --->' rightOperand=CoincidencePattern)*
153 ;
154
155 CoincidencePattern returns gel::DomainSpecificEventsPattern:
156  OrPattern ({gel::CoincidencePattern.leftOperand=current} '&'
       rightOperand=OrPattern)*
157 ;
158
159 OrPattern returns gel::DomainSpecificEventsPattern:
160  XorPattern ({gel::OrPattern.leftOperand=current} '|' rightOperand=
       XorPattern)*
161 ;
162
163 XorPattern returns gel::DomainSpecificEventsPattern:
164  PlusPattern ({gel::XorPattern.leftOperand=current} '><'
       rightOperand=PlusPattern)*
165 ;
166
```

```
167 PlusPattern returns gel::DomainSpecificEventsPattern:
168  IterationPattern ({gel::PlusPattern.operand=current} '+')?
169 ;
170
171 IterationPattern returns gel::DomainSpecificEventsPattern:
172  DomainSpecificEventReferenceOrPattern ({gel::IterationPattern.
       operand=current} '[' numberOfIterations=INT ']')?
173 ;
174
175 DomainSpecificEventReferenceOrPattern returns gel::
       DomainSpecificEventsPattern:
176  DomainSpecificEventReferenceWithOrWithoutTarget
177  | '(' DomainSpecificEventsPattern ')'
178 ;
179
180 DomainSpecificEventReferenceWithOrWithoutTarget returns gel::
       DomainSpecificEventsPattern:
181  DomainSpecificEventReference |
       DomainSpecificEventReferenceWithArguments
182 ;
183
184 DomainSpecificEventReference returns gel::
       DomainSpecificEventsPattern:
185  {gel::DomainSpecificEventReference}
186  referencedDse=[gel::DomainSpecificEvent|ID]
187 ;
188
189 DomainSpecificEventReferenceWithArguments returns gel::
       DomainSpecificEventsPattern:
190  {gel::DomainSpecificEventReferenceWithArguments}
191  referencedDse=[gel::DomainSpecificEvent|ID]
192  '(' arguments=ListOfArguments ')'
193 ;
194
195 ListOfArguments returns gel::ListOfArguments:
196  SingleArgument | MultipleArguments
197 ;
198
199 SingleArgument returns gel::ListOfArguments:
200  {gel::SingleArgument}
201  argument=[gel::LocalVariable|ID]
202 ;
203
204 MultipleArguments returns gel::ListOfArguments:
```

```
205  {gel::MultipleArguments}
206  head=[gel::LocalVariable|ID]
207  ","
208  tail=ListOfArguments
209 ;
210 // End
211
212
213 //-----------------------------------
214 // Miscellaneous elements
215 //-----------------------------------
216
217 // cf. Subsection 3.10.3
218 enum Visibility returns gel::Visibility:
219  public='public' | private='private'
220 ;
221
222 // cf. Section 3.4
223 enum ExecutionKind returns gel::ExecutionKind:
224  submission='triggers' | interruption='interrupts'
225 ;
226 enum CallKind returns gel::CallKind:
227  blocking='blocking' | nonBlocking='nonblocking'
228 ;
229
230 QualifiedName returns ecore::EString:
231  ID (=>'.' ID)*
232 ;
233 // End
```

H

Textual Concrete Syntax of the Projections Metalanguage

As mentioned in Appendix G, there are two approaches to using Xtext [7]. In the first one, the concrete syntax is the main artefact, and Xtext generates a corresponding abstract syntax as an Ecore metamodel. In the second one, the abstract syntax is designed first and then the concrete syntax is specified on top of it. For the Projections metalanguage described in Chapter 4, we used the latter. The abstract syntax (Ecore metamodel) of the language can be found on Figure D.1.

The following listing is the full Xtext textual concrete syntax of the Projections metalanguage.

Listing H.1: The Xtext textual concrete syntax of the Projections metalanguage.

```
grammar org.gemoc.gel.projections.xtext.Projections
with org.eclipse.xtext.common.Terminals

import "http://www.gemoc.org/gel/projections" as projections
import "http://www.eclipse.org/emf/2002/Ecore" as ecore


Projections returns projections::Projections:
        {projections::Projections}
        (imports += ImportStatement)*
        'Projections' ':'
        languageProjections += (LanguageProjection)*
        modelProjections += (ModelProjection)*
        'end'
;

ImportStatement returns projections::ImportStatement:
        'import' importURI=STRING
;

LanguageProjection returns projections::LanguageProjection:
        {projections::LanguageProjection}
        'Language Projection' name = ID ':'
        languageConcept =
          [ecore::EClassifier|QualifiedName] 'projected onto'
        moccConcept = [ecore::EClassifier|QualifiedName]
        'end'
;

ModelProjection returns projections::ModelProjection:
        {projections::ModelProjection}
        'Model Projection' name = ID ':'
        languageElement =
          [ecore::EObject|QualifiedName] 'projected onto'
        moccElement = [ecore::EObject|QualifiedName]
        'end'
;

QualifiedName returns ecore::EString:
        ID (=>'.' ID)*
;
```

Glossary

Hereafter some of the most important terms and their occurrences within this thesis. For most terms, the page number in bold is the (closest thing to a) definition within this thesis's content.

Abstract Syntax (AS) Computer representation of the grammar of a computer language. See pages **25**, 52, 115, 131, 145, 155, 176, 181, 184, 188, 200, 225, 259, 269, 307, 315

Application Programming Interface (API) Software component composed of operations, inputs, and outputs which defines a set of functionalities, independent of its implementation using a particular programming language or for a specific execution platform. See pages 33, 49, 166, 183

Communication Protocol For a concurrency-aware xDSML, language-level specification of the correspondence between the Semantic Rules and the MoCMapping. See pages 48, **62**, 69, 71, 77, 79, 92, 109, 119, 125, 136, 138, 146, 157, 160, 163, 167, 194, 197, 248, 268

Communication Protocol Application For an executable model conforming to a con currency-aware xDSML, model-level specification of the correspondence between the Semantic Rules Calls and the MoCApplication. See pages 49, **64**, 67, 87, 125, 138, 163, 164

Concrete Syntax (CS) Mapping of the Abstract Syntax (AS) of a computer language to a set of rules defining how to parse a string in order to form an instance of the AS of the language. See pages **25**, 184, 188, 307, 315

Concurrency Logical concept related to the dependency that exists (or not) between two pieces of code. See pages vii, xii, 3, 7, **19**, 52, 78, 145, 149, 197

Concurrency-aware xDSML xDSML for which the execution semantics make explicit the systematic use of a MoC for any valid program. See pages viii, xiii, 3, **39**, 48, 108, 117, 140, 145, 154, 172, 176, 181, 184, 194, 197, 225, 259, 269, 289

Domain-Specific Language (DSL) Computer language specialized to a particular application domain. Generally considered in opposition to GPLs. See pages vii, xi, 2, 6, **31**, 33, 174

Domain-Specific Modeling Language (DSML) Computer language specialized to, and presenting adequate abstractions for expressing solutions of, a particular application domain. Generally considered in opposition to GMLs. See pages vii, xi, 2, 6, **37**, 51

Event Structures MoC relying on a set of events constrained by a partial ordering. See pages viii, xiii, 3, **22**, 40, 48, 54, 58, 63, 71, 85, 91, 93, 94, 109, 145, 149, 150, 154, 174, 192, 194, 199, 239, 262

eXecutable Domain-Specific Modeling Language (xDSML) Executable computer language specialized to, and presenting adequate abstractions for expressing solutions of a particular application domain. See pages vii, xii, 2, 6, **38**, 51, 108, 112, 145, 175, 194

Execution Data For a concurrency-aware xDSML, part of the Semantic Rules specifying, as additional attributes and references weaved onto the AS of the language, the dynamic elements evolving during the execution. See pages **56**, 68, 71, 109, 132, 138, 181, 188

Execution Functions For a concurrency-aware xDSML, part of the Semantic Rules specifying, as operations, the evolution of the Execution Data. See pages **56**, 62, 68–70, 72, 76, 78, 83, 92, 98, 104, 109, 112, 118, 132, 140, 188

foundational Subset for Executable UML Models (fUML) Specification by the OMG of an execution semantics for UML Activity Diagrams. See pages 48, **51**, 69, 72, 78, 91, 101, 109, 119, 138, 147, 150, 157, 158, 160, 163, 176, 219, 225, 249, 269, 289

GEMOC International initiative to coordinate research results regarding the development and integration of various modeling languages for the development of heterogeneous systems. See pages 4, **8**, 35, 42, 50, 131, 147, 150, 166, 176, 198, 225, 259, 269, 307

General-purpose Modeling Language (GML) Computer language used to capture models of real-world systems from a wide variety of application domains. Generally considered in opposition to DSMLs. See pages 5, **37**

General-purpose Programming Language (GPL) Computer language used for writing software corresponding to a wide variety of application domains. Generally considered in opposition to DSLs. See pages 5, **31**, 33, 79, 81, 149, 166, 183

Language-Oriented Programming (LOP) Programming paradigm placing languages at the heart of the software engineering activities. See pages vii, xi, 3, **35**, 194, 197

Model of Concurrency (MoC) Formalism used to represent the concurrency concerns of a system. See pages viii, xii, 3, 7, **21**, 48, 58, 71, 74, 92, 140, 145, 149, 154, 174–176, 194, 197, 199

Model of Concurrency Application (MoCApplication) For an executable model conforming to a concurrency-aware xDSML, model-level specification of the use of a MoC to capture the concurrency concerns. See pages 49, **64**, 67, 109, 125, 145, 154, 157, 163, 166, 174, 175, 182, 192

Model of Concurrency Mapping (MoCMapping) For a concurrency-aware xDSML, lan guage-level specification of the systematic use of a MoC for any valid program. See pages 48, **58**, 62, 71, 75, 85, 88, 102, 109, 112, 118, 125, 133, 140, 150, 154, 163, 174, 182, 194, 197, 239, 262

Model-Based Software Engineering (MBSE) Software development methodology based on the use of models, although they are not necessarily the key artefacts of the engineering processes. Superset of MDE. See pages 2, 5, **36**

Model-Driven Engineering (MDE) Engineering paradigm in which models are the key artefacts for the specification, development, testing, validation, verification, etc. of systems. See pages vii, xii, 2, 5, **36**, 51, 166

Object Management Group (OMG) International not-for-profit technology standards consortium, managing modeling standards such as UML, MOF, OCL, XMI and QVT. See pages **36**, 134, 166, 198

Operational Semantics For an executable computer language, a specification of the semantics of the language through a sequence of computational steps. See pages viii, xiii, **28**, 48, 181, 194, 197

Parallelism Physical concept related to the simultaneous execution of two pieces of code (*i.e.*, on two different processors). See pages vii, xii, 3, **19**, 48, 194, 197

Semantic Rules For a concurrency-aware xDSML, language-level specification of the set of data and operations capturing the runtime state and its evolution at runtime. See pages 48, **56**, 62, 69, 71, 75, 92, 98, 102, 125, 146, 155, 162, 194, 197, 200, 225, 269

Semantic Rules Calls For an executable model conforming to a concurrency-aware xDSML, model-level specification of the set of data and operations capturing the runtime state and its evolution at runtime. See pages 49, **64**, 67, 125, 146, 162

Semantic Variation Point (SVP) Part of a specification left intentionally underspecified, allowing implementations to vary in order to cater to different needs while still being conform to the specification as a whole. See pages viii, **28**, 48, 70, 108, 140, 184, 195, 200

Static Semantics For a computer language, additional rules and constraints to its AS, restricting the set of valid programs. See pages **25**, 132, 184

Threads Sequence of programmed instructions. Two variants must be considered: kernel threads, provided and scheduled by the Operating System; and logical threads, provided by a programming language. Both notions are commonly mapped 1:1 although that is not always the case. See pages 21, **22**, 30, 147, 152, 157, 176, 259

Translational Semantics For an executable computer language, a specification of the semantics of the language as a translation to another language with already well-defined semantics. See pages viii, xiii, **28**, 175, 181, 183, 191, 194, 199

Unified Modeling Language (UML) Specification by the OMG of a general-purpose modeling language that intends to provide a standard way to capture the design of a system. See pages 2, 5, 20, 28, **37**, 90, 185

Publications

Weaving Concurrency in eXecutable Domain-Specific Modeling Languages

By Florent Latombe, Xavier Crégut, Benoit Combemale, Julien Deantoni and Marc Pantel.

In 8th ACM SIGPLAN International Conference on Software Language Engineering.

At SLE 2015, co-located with SPLASH and GPCE
in Pittsburgh, Pennsylvia, United States of America.

URL https://hal.inria.fr/hal-01185911

Coping with Semantic Variation Points in Domain-Specific Modeling Languages

By Florent Latombe, Xavier Crégut, Julien Deantoni, Marc Pantel and Benoit Combemale.

In 1st International Workshop on Executable Modeling (EXE 2015), CEUR.

At EXE'15, co-located with MODELS'15
in Ottawa, Canada.

URL https://hal.inria.fr/hal-01222999

Concurrency-aware eXecutable Domain-Specific Modeling Languages as Models of Concurrency

By Florent Latombe, Xavier Crégut and Marc Pantel.

In 2nd International Workshop on Executable Modeling (EXE 2016), CEUR.

At EXE'16, co-located with MODELS'16
in Saint-Malo, France.

URL https://hal.inria.fr/hal-1357001

Don’t we all.

Courtesy of xkcd (http://xkcd.com/1/)

Summary

Language-Oriented Programming (LOP) advocates designing *eXecutable Domain-Specific Modeling Languages* (xDSMLs) to facilitate the design, development, verification and validation of modern software-intensive and highly-concurrent systems. These systems place their needs of rich concurrency constructs at the heart of modern software engineering processes. To ease their development, theoretical computer science has studied the use of dedicated paradigms for the specification of concurrent systems, called *Models of Concurrency* (MoCs). They enable the use of concurrency-aware analyses such as detecting deadlocks or starvation situations, but are complex to understand and master.

In this thesis, we develop and extend an approach that aims at reconciling LOP and MoCs by designing so-called *Concurrency-aware xDSMLs*. In these languages, the systematic use of a MoC is specified at the language level, removing from the end-user the burden of understanding or using MoCs. It also allows the refinement of the language for specific execution platforms, and enables the use of concurrency-aware analyses on the systems.

Résumé

La programmation orientée langage (*Language-Oriented Programming – LOP*) préconise l'utilisation de langages de modélisation dédiés exécutables (*eXecutable Domain-Specific Modeling Languages – xDSMLs*) pour la conception, le développement, la vérification et la validation de systèmes hautement concurrents. De tels systèmes placent l'expression de la concurrence dans les langages informatiques au cœur du processus d'ingénierie logicielle, par exemple à l'aide de formalismes dédiés appelés modèles de concurrence (*Models of Concurrency – MoCs*). Ceux-ci permettent une analyse poussée du comportement des systèmes durant les phases de vérification et de validation, mais demeurent complexes à comprendre, utiliser, et maîtriser.

Dans cette thèse, nous développons et étendons une approche qui vise à faire collaborer l'approche *LOP* et les *MoCs* à travers le développement de *xDSMLs* dans lesquels la concurrence est spécifiée de façon explicite (*Concurrency-aware xDSMLs*). Dans de tels langages, on spécifie l'utilisation systématique d'un *MoC* au niveau de la sémantique d'exécution du langage, facilitant l'expérience pour l'utilisateur final qui n'a alors pas besoin d'appréhender et de maîtriser l'utilisation du *MoC* choisi. Un tel langage peut être raffiné lors de la phase de déploiement, pour s'adapter à la plateforme utilisée, et les systèmes décrits peuvent être analysés sur la base du *MoC* utilisé.