Methods and Tools for the Integration of Formal Verification in Domain-Specific Languages

Faiez Zalila

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *Mardi 09/12/2014* **par :**

Faiez ZALILA

Methods and Tools for the Integration of Formal Verification in Domain-Specific Languages

---

JURY

M. Benoît COMBEMALE	Maître de conférences, IRISA	Examinateur
M. Xavier CRÉGUT	Maître de conférences, INPT	Encadrant scientifique
M. Pierre-Etienne MOREAU	Professeur, École des Mines Nancy	Rapporteur
M. Hassan MOUNTASSIR	Professeur, UFC	Rapporteur
M. Marc PANTEL	Maître de conférences, INPT	Encadrant scientifique
M. François VERNADAT	Professeur, INSA de Toulouse	Président du Jury

---

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (IRIT - UMR 5505)

Directeur(s) de Thèse :
Prof. Yamine AIT-AMEUR

Rapporteurs :
Prof. Pierre-Etienne MOREAU et *Prof. Hassan MOUNTASSIR*

Remerciements

La réalisation d'une thèse est un travail qui nécessite beaucoup d'ambition, d'enthousiasme et de patience. En effet ce travail n'aurait jamais pu être réalisé sans le soutien d'un grand nombre de personnes.

Je tiens à remercier en premier lieu mes chers encadrants Marc Pantel et Xavier Crégut pour la confiance qu'ils m'ont accordée en acceptant d'encadrer mon stage de Master Recherche et ce travail doctoral.

J'étais vraiment chanceux d'être entouré par l'humanité, la gentillesse et la grande culture générale et scientifique de Marc Pantel. Je le remercie infiniment pour sa compréhension, son investissement et sa disponibilité de jour comme de nuit.

Un éternel merci à Xavier Crégut. Merci pour ta disponibilité, ton écoute, ta sympathie, tes conseils et la justesse de tes critiques qui ont guidé mes réflexions et qui ont fait de cette période une formation à la recherche très intéressante et l'objet d'un travail intense. Pour tout cela je vous suis infiniment redevable.

Merci à Yamine Ait-Ameur qui a accepté d'être mon directeur de thèse.

Je remercie Pierre-Etienne Moreau et Hassan Mountassir pour m'avoir fait l'honneur d'être rapporteurs de cette thèse. Je suis honoré pour l'intérêt qu'ils ont porté à ce travail.

Mes vifs remerciements à François Vernadat et Benoît Combemale qui ont bien accepté d'être examinateurs de cette thèse. Je leur exprime ma profonde reconnaissance d'être des membres du jury.

Je tiens à remercier tous les membres du laboratoire IRIT et l'ensemble du personnel de l'IRIT pour leur assistance administrative et logistique ainsi que pour leur sympathie et je pense particulièrement à nos chères secrétaires à Sylvie Eichen, Sylvie Armengaud-Metche et Audrey Cathala.

J'ai une pensée également pour tous les membres de l'équipe ACADIE et toutes les personnes avec qui j'ai partagé mon bureau: Florent, Arnaud, et Ning.

Mes vifs remerciements vont également à tous les amis ceux qui sont en Tunisie et qui m'encouragent toujours, ou que j'ai connu ici en France.

Je dédie cette thèse
à mes parents pour leur soutien et leur encouragement
à ma sœur et son mari pour leurs conseils et aides
et à ma fiancée pour avoir supporté mon stress.

Faiez Zalila

MÉTHODES ET OUTILS POUR L'INTÉGRATION DE LA VÉRIFICATION FORMELLE POUR LES LANGAGES DÉDIÉS

Résumé

Les langages dédiés de modélisation (DSMLs) sont de plus en plus utilisés dans les phases amonts du développement des systèmes complexes, en particulier pour les systèmes critiques embarqués. L'objectif est de pouvoir raisonner très tôt dans le développement sur ces modèles et, notamment, de conduire des activités de vérification et validation (V&V). Une technique très utilisée est la vérification des modèles comportementaux par exploration exhaustive (*model-checking*) en utilisant une sémantique de traduction pour construire un modèle formel à partir des modèles métiers pour réutiliser les outils performants disponibles pour les modèles formels. Définir cette sémantique de traduction, exprimer les propriétés formelles à vérifier et analyser les résultats nécessite une expertise dans les méthodes formelles qui freine leur adoption et peut rebuter les concepteurs. Il est donc nécessaire de construire pour chaque DSML, une chaîne d'outils qui masque les aspects formels aux utilisateurs.

L'objectif de cette thèse est de faciliter le développement de telles chaînes de vérification. Notre contribution inclut 1) l'expression des propriétés comportementales au niveau métier en s'appuyant sur TOCL (Temporal Object Constraint Language), une extension temporelle du langage OCL; 2) la transformation automatique de ces propriétés en propriétés formelles en réutilisant les éléments clés de la sémantique de traduction; 3) la remontée des résultats de vérification grâce à une transformation d'ordre supérieur et un langage de description de correspondance entre le domaine métier et le domaine formel et 4) le processus associé de mise en œuvre.

Notre approche a été validée par l'expérimentation sur un sous-ensemble du langage de modélisation de processus de développement SPEM, et sur le langage de commande d'automates programmables Ladder Diagram, ainsi que par l'intégration d'un langage formel intermédiaire (FIACRE) dans la chaîne outillée de vérification. Ce dernier point permet de réduire l'écart sémantique entre les DSMLs et les domaines formels.

mots clés: Ingénierie dirigée par les modèles (IDM), Langage dédié de modélisation (DSML), vérification et validation (V&V), Object Constraint Language (OCL), vérification formelle, vérification de modèle par exploration exhaustive, sémantique translationnelle, traçabilité, remontée de vérification

---

Institut de Recherche en Informatique de Toulouse - UMR 5505

Faiez Zalila

METHODS AND TOOLS FOR THE INTEGRATION OF FORMAL VERIFICATION IN DOMAIN-SPECIFIC LANGUAGES

Abstract

Domain specific Modeling Languages (DSMLs) are increasingly used at the early phases in the development of complex systems, in particular, for safety critical systems. The goal is to be able to reason early in the development on these models and, in particular, to fulfill verification and validation activities (V&V). A widely used technique is the exhaustive behavioral model verification using *model-checking* by providing a translational semantics to build a formal model from DSML conforming models in order to reuse powerful tools available for this formal domain.

Defining a translational semantics, expressing formal properties to be assessed and analysing such verification results require such an expertise in formal methods that it restricts their adoption and may discourage the designers. It is thus necessary to build for each DSML, a toolchain which hides formal aspects for DSML end-users.

The goal of this thesis consists in easing the development of such verification toolchains. Our contribution includes 1) expressing behavioral properties in the DSML level by relying on TOCL (Temporal Object Constraint Language), a temporal extension of OCL; 2) An automated transformation of these properties on formal properties while reusing the key elements of the translational semantics; 3) the feedback of verification results thanks to a higher-order transformation and a language which defines mappings between DSML and formal levels; 4) the associated process implementation.

Our approach was validated by the experimentation on a subset of the development process modeling language SPEM, and on Ladder Diagram language used to specify programmable logic controllers (PLCs), and by the integration of a formal intermediate language (FIACRE) in the verification toolchain. This last point allows to reduce the semantic gap between DSMLs and formal domains.

keywords: Model Driven Engineering (MDE), Domain specific Modeling Language (DSML), verification and validation (V&V), Object Constraint Language (OCL), Formal verification, Model checking, Translational semantics, Traceability, Verification feedback.

Institut de Recherche en Informatique de Toulouse - UMR 5505

Contents

Introduction

Résumé

Cette introduction présente le contexte général, les défis et la contribution de cette thèse.

Durant la dernière décennie, l'ingénierie dirigée par les modèles (IDM, MDE en anglais) a été exploitée pour améliorer le développement des systèmes critiques embarqués. L'utilisation des modèles dans le contexte industriel améliore le processus de développement car il permet aux utilisateurs de disposer de langages spécifiques de leur domaine donc plus naturels à utiliser que les langages d'implémentation (logiciel, matériel). Cette approche s'appuie sur l'utilisation des langages dédiés de modélisation (DSMLs) qui possèdent des capacités pour décrire un système en utilisant les concepts du domaine considérés.

Cependant, concevoir un DSML est toujours un défi car il nécessite à la fois des connaissances du domaine et l'expertise de développement d'un langage. Un des éléments importants pour définir un DSML est la vérification et la validation (V&V) car les DSMLs sont largement utilisés dans les premières phases du développement de systèmes critiques embarqués. L'utilisation des méthodes formelles pour vérifier de tels systèmes a donné des résultats prometteurs dans le contexte industriel et a suscité l'intérêt des concepteurs de système (les utilisateurs d'un DSML) pour appliquer ces technologies dans des projets industriels réels.

Le coût de développement des outils de V&V est considérable. Par conséquent, il est approprié de s'appuyer sur une sémantique translationnelle qui traduit la syntaxe abstraite du DSML vers un domaine sémantique existant, généralement un langage formel, et permet ainsi de réutiliser les puissants outils (simulateurs, vérificateurs de modèle ou *model-checkers* en anglais) disponibles dans ce domaine.

Cependant, la majorité des concepteurs de systèmes ne maîtrisent pas ces langages formels orientés vérification. Il est donc nécessaire d'intégrer les outils associés dans des chaînes de vérification outillées qui masquent les aspects formels aux concepteurs qui peuvent alors se concentrer sur leurs DSMLs.

L'outil attendu doit remplir plusieurs conditions dont certaines ont été déjà remplies grâce aux technologies de l'IDM comme la définition des modèles en utilisant un éditeur dédié, la vérification de leur conformité au DSML (métamodèle augmenté de contraintes OCL). L'utilisateur du DSML doit également être en mesure de définir les propriétés comportementales en utilisant les concepts de son domaine, puis de les vérifier sur ses modèles. Enfin, l'utilisateur final du DSML veut comprendre les résultats de la vérification, en parti-

culier quand une propriété échoue, sans avoir à plonger dans le côté formel. Ces différents besoins doivent être mis en œuvre pour chaque nouveau DSML. Par conséquent, il est important de faciliter la tâche du concepteur de DSML. Ces concepteurs devraient avoir une telle méthode complète et les outils nécessaires pour intégrer l'activité de vérification facilement pour un nouveau DSML.

La contribution principale de cette thèse vise à faciliter l'intégration de la vérification formelle dans la conception des DSMLs et, plus particulièrement, à donner la possibilité à l'utilisateur final du DSML de vérifier ses modèles sans avoir à se préoccuper des aspects formels et outils associés.

Le premier objectif de notre travail est d'aider les experts et les concepteurs des DSMLs à exprimer des propriétés comportementales au niveau DSML. Pour atteindre cet objectif, notre première contribution consiste à mettre en œuvre une extension temporelle d'OCL correspondant à TOCL. En général, les syntaxes abstraites et concrètes d'un DSML ne contiennent pas tous les éléments nécessaires pour exprimer des propriétés comportementales puisque ces informations apparaissent seulement au cours de l'exécution. Donc, nous devons identifier et modéliser les différentes informations qui seront utilisées lors de l'expression des propriétés comportementales.

Notre deuxième objectif consiste à traduire les propriétés comportementales en propriétés du domaine formel. Nous fournissons une transformation de modèle d'ordre supérieur (HOT) qui engendre une transformation de modèle produisant les propriétés formelles correspondantes. Ces transformations s'appuient sur la sémantique translationnelle utilisée.

Le troisième objectif consiste à aider le concepteur du DSML à interpréter les résultats de vérification obtenus au côté formel. Notre but est de fournir une solution générique qui peut être appliquée sur tout DSML et tout langage formel et qui est indépendante de la façon dont la sémantique translationnelle a été codée. Nous fournissons un langage de programmation dédié (DSPL), nommé FEVEREL (Feedback Verification Results Language) qui permet de définir des correspondances entre les informations d'exécution du niveau DSML et celles du niveau formel. Ensuite, nous proposons une transformation d'ordre supérieur (HOT) qui génère automatiquement une transformation de modèle correspondante qui transforme les résultats de la vérification vers le niveau métier.

Le quatrième objectif de cette thèse concerne le côté méthodologique de définition et d'utilisation d'une chaîne outillée de vérification d'un DSML. Il est recommandé que l'intégration peut se faire d'une manière bien structurée. Ceci permet par exemple d'identifier quel type d'information doit être mis à jour lorsque le domaine formel est substitué par un autre ou lorsque la sémantique translationnelle est mise à jour.

Notre dernière contribution consiste à récapituler les différents éléments de l'activité de vérification pour un nouveau DSML. Ceci fournit une vue de haut niveau sur l'intégration de la vérification formelle pour un nouveau DSML. Elle identifie la manière dont le concepteur d'un DSML se comporte quand il choisit de changer une telle partie de la chaîne outillée de vérification. En outre, le concepteur d'un DSML a encore des difficultés à intégrer la vérification formelle en raison de l'écart sémantique entre les DSMLs et les sémantiques des domaines formels. Par conséquent, sur la base de la méthode proposée, nous avons dé-

cidé de valider notre approche par l'intégration d'un langage formel intermédiaire dans la chaîne de vérification outillée afin de réduire cet écart sémantique.

0.1 Context and challenges

In the last decade, Model Driven Engineering (MDE) has been used to improve the development of safety critical systems. The use of models in the industrial context improves the current development process for experts and users by creating rigorous models and thus reducing the costs. Indeed, the MDE aims to provide languages close to users domains and easier to use than implementation ones (software, hardware). This approach relies on the use of *Domain specific Modeling Languages* (DSMLs) that have the capabilities to describe a system using its domain concepts.

However, designing a DSML is still a challenging and time-consuming task because it requires both domain knowledge and language development expertise. To design a DSML, the domain expert explains different requirements that should be achieved. Based on these requirements, software language designers must implement different DSML concerns like the abstract syntax, the concrete syntax and the DSML semantics. Finally, the domain expert should validate whether DSML requirements are respected by software language designers [CGS12].

One of these DSML requirements is model verification and validation (V&V) because DSMLs are widely used in the early phases of the development of safety critical systems. These activities are key features to assess the conformance of the future system to its safety and liveness requirements. Verification activity based on formal methods of safety critical embedded systems has produced very promising results in the industrial context and raised the interest of system designers (DSML end-users) up to the application of these technologies in real size projects [BVWW09, WL03, Lec09].

As an example, TOPCASED[1] is a research and development project started in 2005 in the French "Aerospace Valley" cluster that gathers academic and industrial partners [FGC+06]. It is dedicated to the development of open source Computer Assisted Software Engineering (CASE) toolset for the development of safety critical aeronautic, automotive and space embedded systems. Such developments will range from system and architecture specifications to software and hardware implementation through equipment definition.

TOPCASED addresses modeling languages, both domain specific ones (SAM, EAST-ADL, AADL, and SDL[2]) and general purpose ones (SYSML, UML, etc.) and associated tools like graphical and textual editors, documentation generators, validation through model animation, verification through model checking, version management, traceability, etc.

As the cost of developing new V&V tools is significant, it is appropriate to introduce a translational semantics for DSMLs which is provided as a mapping from the abstract syntax (metamodel) of the DSML to an existing semantic domain, usually a formal language, in order to reuse powerful tools (simulator or model-checker) available for this domain [MP10, HR04].

However, most system designers do not master these specific verification-oriented formal languages. It is thus mandatory to embed the associated tools in automated verification

[1] *Toolkit In OPen source for Critical Applications & SystEms Development*, `www.topcased.org`

[2] *Specification and Description Language: is an object-oriented formal language developed and standardized by The International Telecommunication Standardization Sector (ITU-T)*

toolchains that allow designers to focus on their usual DSMLs, hiding all formal aspects but still enjoying the benefits of the powerful tools.

The expected tool has to fulfill several requirements. Some are already achieved thanks to MDE technologies: defining models using a dedicated editor and checking its conformance to the DSML as well as to OCL constraints. The DSML end-user must also be able to define behavioral properties using the concepts of its domain and then to verify whether these properties hold or not [on the models]. Finally, the DSML end-user wants to understand verification results, when a property fails, without having to dive in the formal side.

These different requirements should be implemented for each new DSML. Therefore, it is important to ease the DSML designer task. DSMLs designers should have such a complete method and the necessary tools to integrate easily verification activity for a new DSML.

0.2 Description of the thesis contributions

Our global thesis contribution aims to ease the integration of the formal verification in the design of DSMLs and, more particularly, it consists in giving the possibility for the DSML end-user to verify its models without having to deal with formal aspects and their related tools underlying the verification activity.

The first goal of our work is to help DSML expert and designers to express behavioral properties at the DSML level and their related elements. To achieve this objective, our first contribution consists in implementing a temporal extension of Object Constraint Language (OCL) corresponding to TOCL as proposed by Paul Ziemann and Martin Gogolla in [ZG02] that allows the DSML expert and designer to express the behavioral properties to assert and their related elements. Usually the DSML abstract and concrete syntaxes do not contain all necessary elements to express behavioral properties as they relate to the information existing only during the execution which is not most of the time modelled by the abstract syntax. So, we need to identify different kind of information that should be added during the expression of the behavioral properties.

Our second goal consists in managing the expressed behavioral properties. We provide a higher-order model transformation (HOT) that generates a model transformation producing the corresponding formal properties. So, we explain our proposed translation to automatically generate formal properties and we stress the elements on which this translation relies.

The third objective consists in assisting the DSML designer to manage verification results obtained on the formal side. Our purpose is to provide a generic solution which can be applied on any DSML and any formal domain and which is independent of how the translational semantics was coded. We provide such a domain-specific programming language (DSPL), named FEVEREL (Feedback Verification Results Language) that allows to define a mapping between the DSML runtime information and the formal one. Then, we provide a higher-order transformation (HOT) that generates automatically a corresponding model transformation which transforms verification results from the formal side into the DSML one.

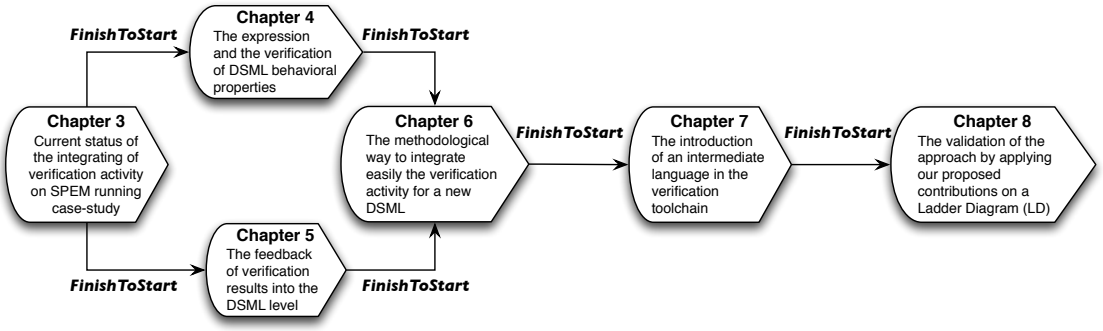

Figure 1 — Towards a generic approach to integrate formal verification for DSMLs

The fourth objective concerns the methodological side of defining and using the verification toolchain for a new DSML. It is recommended that the integration can be done in a well-structured way. It allows for example to identify what kind of information should be updated when the formal domain is substituted by another one or when the translational semantics is updated. Our last contribution consists in summarizing different verification activity elements. It provides a high level view of the integration of formal verification for a new DSML. It identifies how the DSML designer behaves when he chooses to change such a part of the verification toolchain.

Furthermore, the DSML designer still has difficulties to integrate formal verification due to the semantic gap between DSMLs and formal semantics domains. Therefore, based on the proposed method, we decided to validate our approach by integrating a formal intermediate language in the verification toolchain in order to reduce this gap.

0.3 Outline of this thesis

This part gives a brief summary of this thesis which is composed of 8 chapters and structured into 3 parts:

- **Part 1: State of the Art**
  - Chapter 1 introduces the technical background related to the modeling world by presenting the model-driven engineering (MDE), the model driven architecture (MDA) and model transformations.
  - Chapter 2 presents the notion of domain-specific modeling language (DSML), the different required elements (abstract syntax, concrete syntax and semantics) to build it and the related verification activities.
  - Chapter 3 explains the running case-study which is considered as the pivot case-study of our work during this thesis. It relies on the Software Process Engineering Metamodel (SPEM). We present the proposed verification activity. Based on this approach, we discuss the missing elements to obtain a seamless approach to ensure the verification activity for DSMLs.

- **Part 2: Contribution**
  - Chapter 4 handles the first identified problematic which is **the expression and the verification of behavioral properties**. We show our proposed language to express behavioral properties at the DSML level. Then we explain our proposed translation to automatically generate formal properties.
  - Chapter 5 deals with **the feedback of verification results** problematic. It introduces our proposed language to manage verification results (FEVEREL) and the proposed solution to transform formal verification results into DSML ones.
  - Chapter 6 represents from **a methodological viewpoint, the integration of the verification activity for a new DSML** and explains how to obtain a DSML verification framework. It stresses the dependencies between the different verification activity parts and details the variant and invariant aspects when such an element in the verification activity toolchain changes.
- **Part 3: Validation of the approach**
  - Chapter 7 introduces **an intermediate formal language in the verification toolchain** to reduce the semantic gap between DSMLs and formal languages. We apply the methodology presented in the previous chapter, by substituting the formal target language, to show the generic aspects of our approach.
  - Chapter 8 validates our approach by **applying our proposed contributions on a DSML named Ladder Diagram (LD)** used to model Programmable Logic Controllers (PLCs). It consists in formalizing generic properties at the LD level and feeding back verification results at the LD level in order to be understood by domain engineers.

Figure 1 shows a process model that describes the principal contributions of this thesis. Dependencies between activities correspond to the possible paths for reading this manuscript. Finally, we conclude this thesis and we outline future directions for research.

Part

State of the Art

1 Model-driven Engineering

Résumé

Ce premier chapitre présente le cadre théorique et technique de cette thèse. Il détaille les notions clés de l'IDM.

Durant la dernière décennie, l'IDM a été utilisée pour améliorer le processus de développement des logiciels en réduisant la complexité des différentes phases de développement, en élevant le niveau d'abstraction dans la spécification d'un programme et en permettant les activités de V&V dans les phases amont. L'IDM est appliquée avec succès dans de nombreux domaines comme l'automobile et l'aéronautique. L'idée principale de l'IDM consiste à considérer les modèles comme l'artefact principal pour le développement des systèmes. Un *modèle* est une vue abstraite d'un système qui permet de comprendre le système modélisé et répondre à des questions connexes. Il est défini conformément à un *métamodèle* qui introduit un métalangage permettant d'exprimer des modèles. La définition d'un métamodèle est le processus de métamodélisation (c'est-à-dire la définition d'un langage).

En 2001, le consortium international *Object Management Group* (OMG) a normalisé l'IDM et a proposé l'approche *Model Driven Approach* (MDA) comme une méthode pour appliquer l'IDM. L'approche MDA est fondée sur la séparation des préoccupations. Elle permet de modéliser séparément les aspects métiers et techniques d'un système. Cette initiative vise à normaliser l'utilisation de modèles en fournissant un ensemble d'outils et de méthodes comme MetaObjectFacility (MOF), Unified Modeling Language (UML), XML Metadata Interchange (XMI), Object Constraint Language (OCL), etc. L'approche MDA repose sur une architecture de métamodélisation à quatre niveaux. Un premier niveau, M0, nommé aussi le niveau d'instance, correspond au monde réel. Il décrit le système concret. Ce dernier est représenté sous forme de modèles au niveau M1 (le niveau modèle). Ces modèles sont conformes à leurs métamodèles du niveau M2. Un métamodèle définit un domaine de connaissance. Ces métamodèles eux-mêmes sont conformes au méta-métamodèle MOF (niveau M3) qui est un métamodèle décrivant un langage de métamodélisation.

Un des processus importants dans le contexte de l'IDM est la transformation de modèle. Elle permet d'automatiser la manipulation des modèles et consiste à produire un modèle cible à partir d'un modèle source (on dit M2M, modèle à modèle) conformément à une définition de transformation. Dans ce chapitre, nous présentons une classification des transformations de modèle en nous appuyant sur la nature des métamodèles de la transformation

(transformations exogènes ou endogènes) et le niveau d'abstraction des modèles manipulés (transformations verticales ou horizontales). Un cas particulier de transformation est la transformation de modèle à texte (M2T) (génération de code, documentation, etc). Une transformation est elle-même un modèle et peut être l'entrée ou le résultat d'une transformation, cette dernière est dite transformation d'ordre supérieur (HOT). À la fin de ce chapitre nous citons quelques exemples de langages de transformation de modèles : ATL, Kermeta et QVT.

In the last decade, Model Driven Engineering (MDE) has been used to improve the software development process by reducing the complexity of different development phases, by raising the level of abstraction in the program specification and by introducing early V&V activities. MDE is applied successfully in many domains like automotive and aeronautics.

The principal idea in MDE consists in considering models as the main artifact for developing systems. A *model* is an abstract view of a system which allows to understand the modelled system and answer to related questions. It is defined in conformance to a *metamodel* which defines a language enabling to express models [Béz06]. Defining a metamodel is the process of metamodeling (i.e. language definition).

In 2001, the Object Management Group's (OMG) standardized the MDE and proposed the MDA approach as a method for applying MDE.

One of the most important processes in the MDE context is the model transformation. It consists in producing a target model from a source model conforming to a transformation definition [MVG06].

In this chapter, we present different notions of the MDE. First, we introduce the notion of model, metamodel, and metamodeling (section 1.1). Then, we show the MDA approach proposed by the OMG and its architecture (section 1.2). Section 1.3 defines the concept of model transformation and its different kinds. In addition, we show existing tools for model transformation.

1.1 Model and Metamodel

Since the sixties, Object technologies are based on the basic principle *"Everything is an object"*. It has provided more simplicity, generality and power of integration of this technology for which two core relations are identified: the inheritance (*inheritsFrom*) and the instantiation (*instanceOf*). This direction has been followed when the MDE appeared with the basic principle (*"Everything is a model"*) [Béz05]. The MDE aims at increasing the abstraction level in the development process by the use of models in the different development phases. In the MDE, the notion of a *model* is the core of the development.

Several definitions of the notion of *model* can be identified in the literature. Minsky in [Min68] proposed the following definition: *« To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A.»* In [BG01], another definition of a *model* was proposed. We consider below this definition.

Definition 1. *A model is a representation or an abstraction of a (part of a) system. It can be used, instead of the real system, to answer questions that can be asked about this system.*

Based on this definition, a first principle for the MDE was identified [Béz04]. It is the representation (*representedBy*) relation between a system and a model (the bottom of Figure 1.1). Once we choose to represent a system with a model, it is mandatory to specify how we can define a model. It is done through a language which obviously is a model, called *metamodel*.

Definition 2. *A metamodel is a model that defines a language to specify conforming models [OMG06]. It is thus a modeling language.*

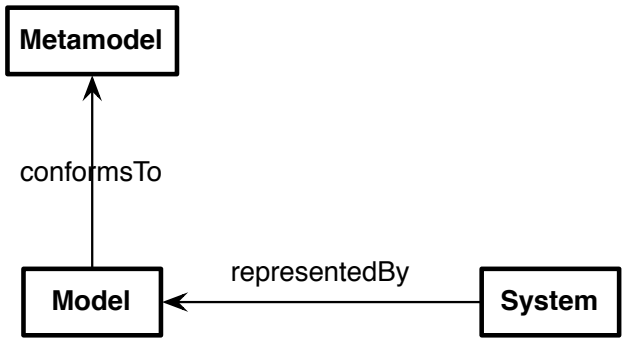

Figure 1.1 — MDE core relations

A metamodel allows to formalize a domain, its concepts and the relations between them. The metamodel becomes the core of the different development phases for this domain [JCV12].

The notion of metamodel allows to identify a second kind of relation in the MDE context between a model and a metamodel. It is the conformance (*conformsTo*) relation shown vertically in Figure 1.1. A model conforms to its metamodel.

1.2 Model-driven Architecture

In 2001, the OMG launched a software design approach for MDE named model driven architecture (MDA) [OMG03a]. The MDA approach is based on the seperation of concerns. It allows to separately take into account business and technical aspects of a system due to the modeling process. This initiative aims to standardize the use of models by providing a set of tools and methods.

1.2.1 The MDA approach

The MDA can be defined as the OMG vision for application of the MDE. It consists in defining a software framework to use models in the software development. Therefore, several standards have been proposed in this approach like:

- The Meta Object Facility (MOF) provides the elementary constructs to define metamodels, to extend or to modify existing ones. It conforms to itself [OMG06].
- The Unified Modeling Language (UML) is a general purpose modeling language (GPML). It was proposed as a graphical modeling language for the design of a software system. It is an object oriented modeling language that includes a set of graphical notations to design the structural and the dynamic views of a system [OMG07b]. It has been extended to provide more than 14 different kinds of diagrams (for UML 2.3 [OMG10]) and can be further extended with the profile reflexive facilities. This format

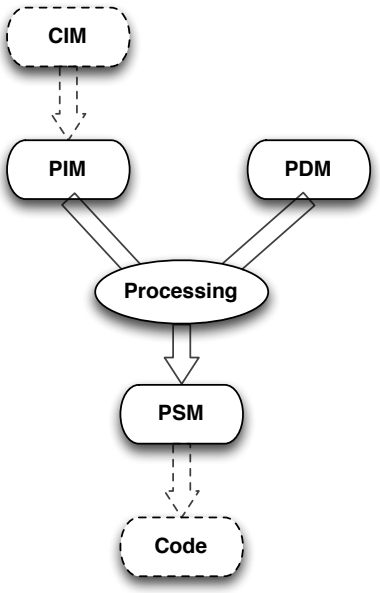

Figure 1.2 — The *Y* schema of the MDA approach

has been extended by UML-DI (Diagram Interchange) to embed graphical data related to diagrams.

- The XML[1] Metadata Interchange (XMI) is a standard for exchanging metadata information via XML. It complements the UML modeling languages by defining an interchange format based on XML. The XMI ensures the interoperability and serialization techniques for models [OMG11b].
- The Object Constraint Language (OCL) is a textual constraint language that completes the specification which may be ambiguous due to the graphical notation of modeling languages [OMG12].

The main goal of the MDA is to separate different system considerations during the development process. For instance, it aims at distinguishing between the specification and the implementation of a system in order to ease the maintenance. As shown in Figure 1.2, several types of models can be identified:

- *Computational Independant Model* (CIM): defines the requirements that describe functional needs for an application.
- *Platform Independent Model* (PIM): represents the design of the system without any implementation consideration. It allows to give a structural and a dynamic view of the system, always regardless of any technical design of the system.
- *Platform Description Model* (PDM): specifies the platform model of the implementation (J2EE, .Net, PHP, etc.).

[1] eXtensible Markup Language

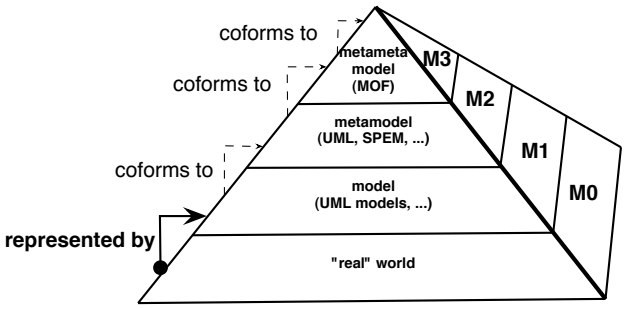

Figure 1.3 — MDA layers

- *Platform Specific Model* (PSM): is the closest model to the code. It can be the result of combining the PIM with the PDM.

The code, in the MDA approach, is usually automatically generated from different models that represent a system. They are not only a visual way to ease the understanding of the application but also a productive and pivot element in the MDA process.

1.2.2 The MDA architecture

The MDA is based on the four-layer metamodeling architecture as shown in Figure 1.3. The $M0$ layer, named also instance level, corresponds to the real world. It describes the concrete system. It is abstracted as models in the $M1$ layer (the model level). These models conform to their metamodels given in the $M2$ layer. A metamodel defines a knowledge domain. These metamodels conform to the MOF metametamodel ($M3$ layer).

Definition 3. *A metametamodel is a metamodel that describes a metamodeling language. It provides a set of constructs that allow to define modeling languages. It conforms to itself.*

Figure 1.4 shows a concrete modeling example conforming to the four layers of the MOF architecture. It illustrates this architecture by modeling the file system. The bottom shows a real file system as observed by the user. It represents the real world. This file system can be abstracted as a model which is proposed in the $M1$ layer: the model layer. This model conforms to a metamodel that defines the concepts of this domain. It introduces the concept of *Filesystem* which represents the whole system, a set of notions like *Drive*, *File*, etc. and the relations between them like composition and inheritance. This metamodel conforms to the MOF metametamodel. Dashed arrows shows the conformance relation between the model and the metamodel on the one hand and between the metamodel and the MOF metametamodel on the other hand.

The MOF standard offers elementary constructs which allow to describe metamodels. There are a lot of frameworks aligned on OMG's MOF: Eclipse-EMF/Ecore [BSE03], AMMA/KM3 [JB06] or Kermeta [MFJ05]. These languages have the required concepts to define new metamodels. For instance, they provide constructors for structural elements (*Class*). A

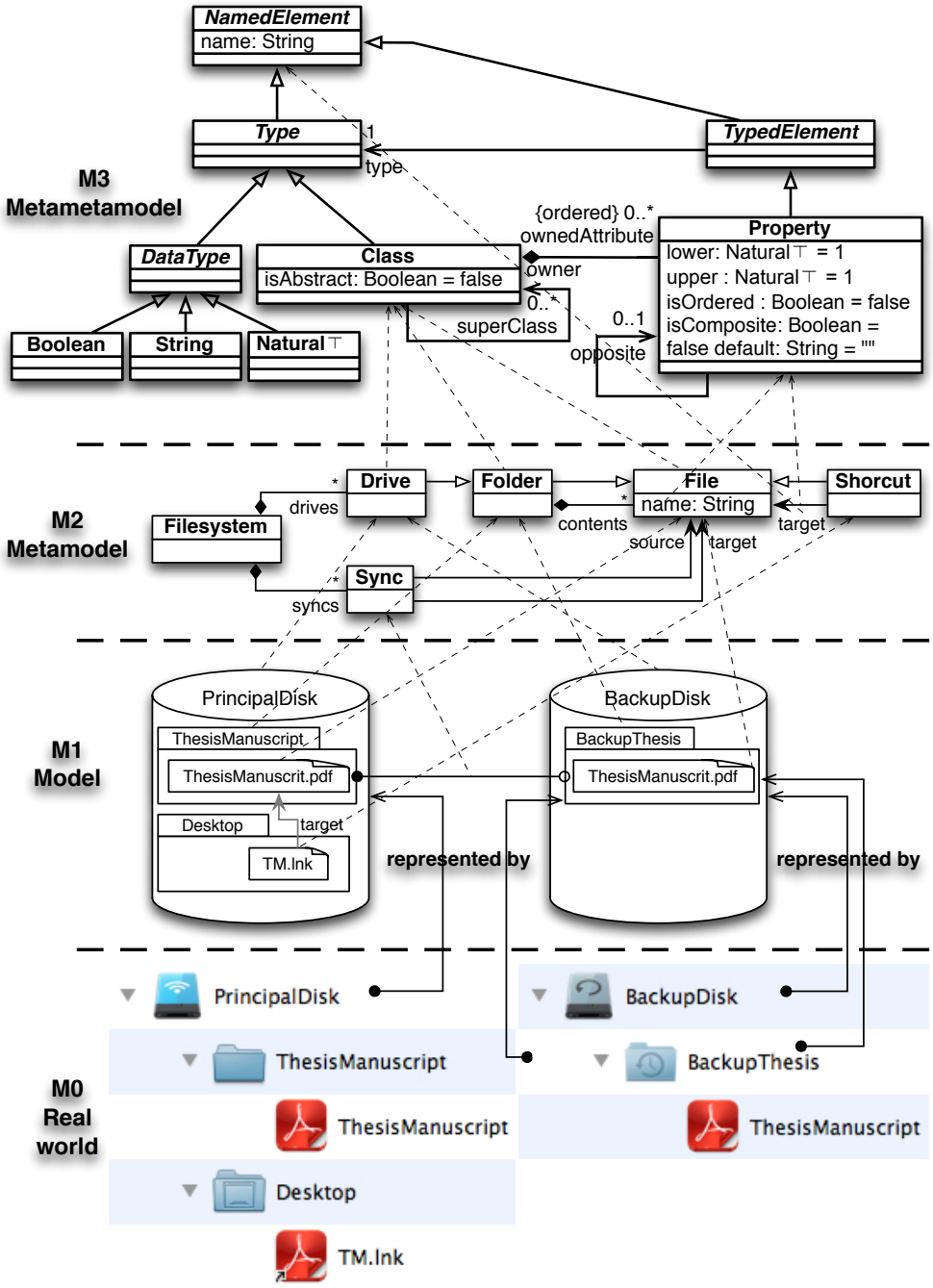

Figure 1.4 — A concrete modeling use-case

Class is composed of characteristic properties (*Property*). A property is considered as a *reference* if it is typed by another class (*TypedElement*) and an *attribute* when it is typed by a primitive type (*Boolean*, *String* or *Natural*).

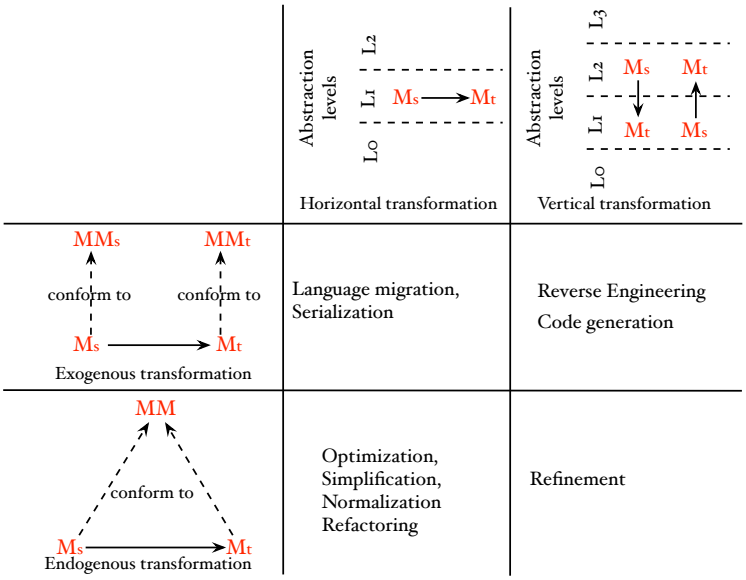

Figure 1.5 — Model transformation types and their main uses

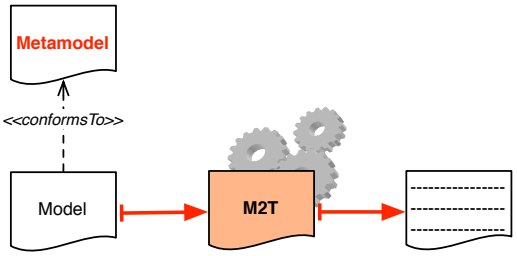

Figure 1.6 — Model-to-text transformation

1.3 Model Transformation

The MDE considers the "model" notion as a key artifact and the core of the development process. So, it is necessary to ease the use of the defined models. Model transformation is a central concept in the MDA approach. It provides a mechanism to automate the manipulation of models. It is considered as programs that take models as inputs and build new models as outputs.

In this section, we describe the various kinds of model transformations (subsection 1.3.1). Then, we present, briefly, some model transformation languages used by the MDE community.

1.3.1 Model transformation types

A *model-to-model* (M2M) transformation is the generation process of a target model (M_t), conforming to a target metamodel (MM_t), from a source model (M_s) conforming to a source metamodel (MM_s).

In the literature, there are many proposed criteria to classify model transformations. One of the classification criteria is the nature of the transformation metamodels. Two kinds of model transformation can be identified in this field:

- *exogenous model transformation*: where the input and output models conform to different metamodels. This kind of transformation allows to migrate from a model written in one language to another (*language migration*). In addition, this kind of transformation can synthesize a high-level specification into a lower-level. This use corresponds to the *code generation* process where the design models are translated into the source code. Furthermore, this transformation kind eases extracting a higher-level specification from a lower-level one (*Reverse engineering*).

- *endogenous model transformation*: where the input and output models conform to the same metamodel. MM_s and MM_t are the same. This kind of transformation has several utilities. For instance, it aims to optimize the performance of a model while preserving its semantics (*Optimization*). In addition, it can improve the internal structure of the software in order to improve its quality characteristics without changing its external observable behavior (*Refactoring*). Another purpose of the endogenous transformation is the *simplification* and the *normalization* which mean decreasing the syntactic complexity of a model. Finally, this kind of transformation can refine an abstract specification into a more concrete specification (the *refinement*).

Another kind of classification criteria can be studied is the abstraction level of different models manipulated during a transformation. Two kinds of model transformation are identified:

- A *vertical transformation* is a transformation where the source and target models belong to two different levels of abstraction. A typical example is the *code generation* where the abstraction level decreases during this process.

- A *horizontal transformation* is a transformation where the abstraction levels of the source and the target model are the same. A typical example is the *refactoring*.

Figure 1.5 shows an orthogonal classification of model transformation based on both cited classification criteria.

There are many other classification criteria for model transformations like the supported target type (a transformation which allows generating texts from source models (Figure 1.6) is named *model-to-text* (M2T) transformation) and the directionality of a transformation which can be unidirectional (only from source to target) or bidirectional (a transformation can be applied from source to target and from target to source). Several studies are proposed in [Bie10, MVG06] to list different model transformation classification criteria.

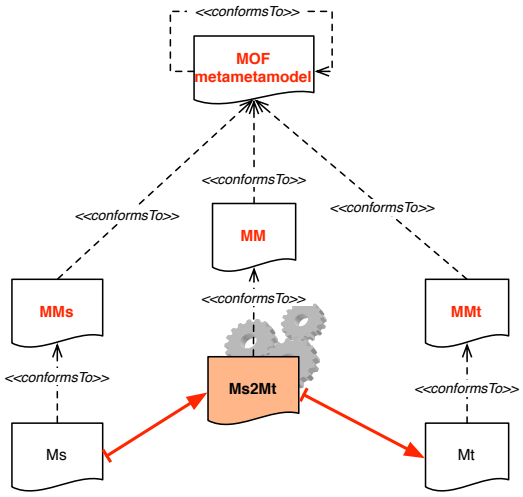

Figure 1.7 — Model transformation process

Model transformations have been used in many different domains and their popularity is growing due to their increasing success to handle complex applications and processing in these domains. The evolution of the MDE is characterized by considering model transformations as an integral part of the developed system. So, model transformations can be themselves generated as traditional programs. Due to the basic principle of MDE ("*Everything is a model*"), a new concept was proposed: transformation model. A transformation model can be created, modified, extended via a transformation. The transformation model conforms to a transformation metamodel (Figure 1.7) which conforms to the MOF metametamodel [BBG+06]. Considering model transformations as models eases their manipulation using model transformations named *Higher-Order Transformations* (HOT).

Definition 4. *A higher-order transformation is a model transformation that manipulates other model transformations. It means that the input and/or output models are themselves model transformations.*

1.3.2 Model transformation languages

Since the appearance of the MDE and the MDA, many model transformation languages have been proposed. First, there are generic model transformation languages like the EMF API [2] for Java where the model transformation is coded as a Java program. Then, a set of specific model transformation languages are proposed such as:

Kermeta [MFJ05] is defined as a meta-modeling, object-oriented and aspect-oriented programming language. It uses EMF tools to define programs which are also models, to specify transformations of models, to specify constraints on these models, and to execute them.

ATL (Atlas Transformation Language) is a hybrid transformation language (declarative and imperative). The declarative style of ATL allows to simplify complex transfor-

[2] Application Programming Interface

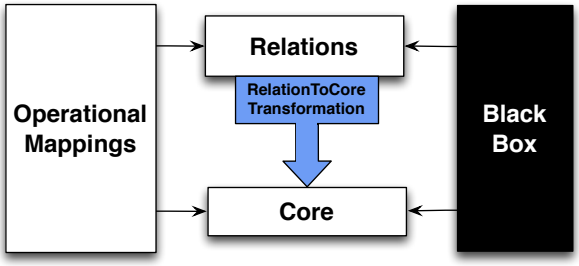

Figure 1.8 — QVT standard architecture

mations algorithms. The imperative constructs allow to specify mappings that are too hard to be defined declaratively. ATL allows to define a model-to-model transformation (named *Module*) and model-to-text transformation (named *Query*). An ATL model-to-model transformation is composed of rules that define how source model elements are handled to create and to initialize the elements of target models. ATL is defined both as a metamodel and as a textual concrete syntax [JABK08].

QVT (Query/View/Transformation) is the OMG standard language for specifying model transformations [OMG11a]. The QVT metamodel conforms to the MOF. It uses the Object Constraint Language (OCL) to navigate on models. QVT defines three transformation languages: (1) QVT *Relations* which is a declarative model transformation language. It supports the specifications of bidirectional model transformations. (2) QVT *Core* which is a low-level declarative model transformation language. It provides a foundation for the QVT *Relations* semantics which is defined as a transformation from *Relations* to *Core*. (3) QVT *Operational Mappings* which is an imperative model transformation language. It extends the *Relations* with imperative constructs. The specified transformations are unidirectional. Finally, the QVT architecture has a mechanism called *Black Box* for invoking transformation facilities expressed in other languages (Figure 1.8).

Many other model transformation languages exist. We give some and their references: ETL (Epsilon Transformation Language) [KPP08], VIATRA2 [VB07], Tom [BCMP12], etc.

2 Domain-specific Modeling Languages

Résumé

Dans le contexte de l'IDM, les modèles jouent un rôle prédominant durant le processus de développement. Par conséquent, il est naturel de formaliser la définition d'un domaine en terme de ses concepts et des relations entre eux sous forme d'un métamodèle, qui représente une vue de haut niveau du monde réel. La métamodélisation consiste à définir un métamodèle (nommé également un langage de modélisation) qui représente l'ensemble des modèles conformes à ce langage.

Il y a deux paradigmes qui guident le développement de ces langages de modélisation: la modélisation généraliste (general-purpose modeling en anglais (GPM)) et la modélisation dédiée à un domaine (domain-specific modeling en anglais (DSM)). La GPM consiste à utiliser un langage de modélisation généraliste (general-purpose modeling language en anglais (GPML)) pour représenter plusieurs aspects d'un système sous forme d'un modèle. Un GPML est un langage de modélisation (et éventuellement son outillage) qui peut être appliqué à n'importe quel domaine. UML est un exemple typique de GPML utilisé pour modéliser une grande variété de systèmes. La DSM est une méthode de génie logiciel pour concevoir et développer des systèmes en s'appuyant sur de nombreux modèles différents correspondants aux différents aspects d'un système. Elle consiste à utiliser un langage dédié à un domaine (domain-specific language en anglais (DSL)) pour définir (une partie d') un système. Un langage de modélisation dédié (domain-specific modeling language en anglais (DSML)) est un DSL utilisé pour modéliser tels systèmes. Contrairement à un GPML, un DSML capture les concepts d'un domaine spécifique.

L'utilisation de DSML a bien mérité sa place intéressante dans la communauté de génie logiciel. Il inclut des concepts de haut niveau qui correspondent aux termes réels de l'utilisateur final du domaine.

Pour concevoir un DSML, trois éléments de base devraient être définis: une syntaxe abstraite qui représente la structure du langage, des syntaxes concrètes (textuelles ou graphiques) qui décrivent des représentations spécifiques du DSML et plusieurs sémantiques qui fournissent le sens des éléments définis au langage. Comme nous ciblons les systèmes critiques embarqués, les activités de vérification et validation (V&V) sont essentielles. Nous devons fournir des outils de haute qualité pour les utilisateurs finaux des DSMLs. La vérification de modèle est une étape critique dans le processus de développement. Elle

consiste à évaluer la conformité des modèles conçus en s'appuyant sur la sémantique du DSML.

In the MDE context, models play a dominant role during the development process. Hence, it was natural to define more abstract models, named metamodels, that represent high-level views of the real worlds. Metamodeling defines a metamodel (named also a modeling language) that represents the whole class of models conforming to this language [BCW12].

There are two paradigms that guide the development of such modeling languages: *general-purpose modeling* (GPM) and *domain-specific modeling* (DSM).

GPM consists in using a *general-purpose modeling language* (GPML) to represent multiple aspects of a system as a model. A GPML is a modeling language (and eventually its tooling) which can be applied to any domain. UML is a typical example of GPML used to model a wide variety of systems.

The DSM is a software engineering methodology to design and develop systems relying on many different models corresponding to the various aspects in a system. It consists in using a domain-specific language (DSL) to define a (part of a) system. DSLs have been widely used in the different computer science fields like HTML[1] markup language for Web page development, SQL[2] to query databases, etc. A *domain-specific modeling language* (DSML) is a DSL used to model such systems. In contrast to GPML, DSML captures the concepts of a specific domain.

The use of DSML has well earned its interesting place in the software engineering community. First, a GPML may only provide generic modeling concepts far from the end-user domain ontology. In addition, it may provide all the potential modeling concepts and overwhelm the end-user that will have many different ways of modeling the same artifact. However, a DSML includes high-level concepts that correspond to the real terms of the user domain. In addition, a DSML is developed with its specific graphical or textual syntax which is more near to the user knowledge and with its specific constraints that check the validity of defined models [Fra11].

As we target the development of tools for safety critical systems, the reliability of the designed tools is crucial. To ensure the suitability of the DSMLs and their related tools, it is necessary to introduce the verification technique to the developed modeling languages. It consists in checking whether DSML conforming models behave as expected.

In this chapter, we detail different required elements to design a DSML. We illustrate them on a DSML for describing processes based on the *Software Process Engineering Metamodel (SPEM)* [OMG07a]. Then, we explain different approaches proposed to perform model verification in the DSML context.

2.1 Different elements defining a DSML

To design a DSML, three core elements should be defined: an abstract syntax that represents the structure of the language, concrete syntaxes that describe specific representations of the DSML and several semantics that provide the meaning of the elements defined in the

[1] Hypertext Markup Language
[2] Structured Query Language

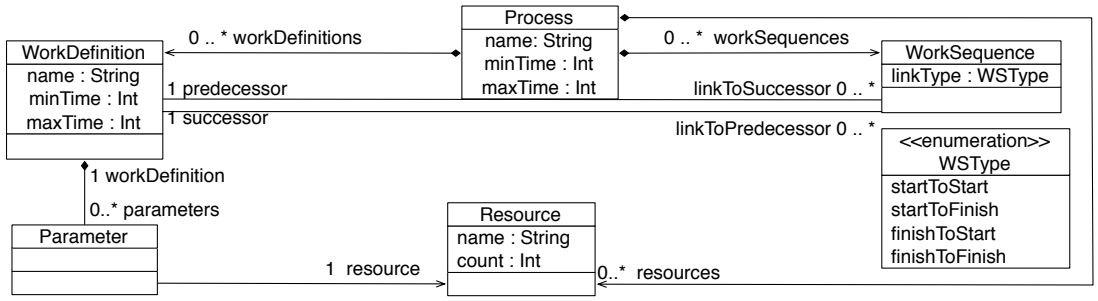

Figure 2.1 — An extract of SPEM

language and must be consistent.

2.1.1 Abstract syntax of a DSML

An abstract syntax defines the structure of a language, the whole language concepts and their relationships. It is defined using a metamodeling framework, based on the MOF metametamodel, like the Eclipse-EMF/Ecore.

Many abstract syntaxes currently available also targets other uses like being the inputs for other tools and thus can suffer from other requirements and thus get away from the initial purpose of giving a simple and minimal definition of domain concepts and relations. As a well known example, the UML metamodels suffer from many requirements like interchange formats, diagrams supports, factorisation, etc.

SPEM is an OMG standard defined in order to specify and describe software and system development processes. A subset of the SPEM 2.0 is shown in Figure 2.1. It defines the concepts of *Process* composed of (1) a set of activities (*WorkDefinition*) performed during the process, (2) a set of dependencies (*WorkSequence*) that define temporal dependency relations (causality constraints) between activities and (3) a set of resources (*Resource*) allocated to activities (*Parameter*).

WorkDefinitions are related thanks to the *WorkSequence* concept, whose attribute *linkType* specifies when an activity can be started or finished. The values of kind are defined by the *WSType* enumeration. A WorkSequence value follows the *stateToAction* pattern (*startToFinish* type means that the target activity can only finish when the source activity has been started).

As for the majority of modeling languages and due to the lack of expressivity of the graphical representation of metamodeling languages, the proposed SPEM metamodel does not capture the whole DSML requirements. For example, the requirement "workdefinition names have to be unique within a process" cannot be captured by the SPEM metamodel. Therefore, the DSML metamodel should be extended with well-formedness properties which must be respected by the conforming models. OCL [OMG12] is the OMG standard proposed to define such properties on models. It is a general-purpose textual formal language. OCL constraints are first order logic formulas defined as invariants for each spe-

cific type associated to metaclasses (*context*). Its library defines the primitive and collection-related types and their predefined operations. In addition, OCL has an universal quantifier *forAll* and an existential quantifier *exists* and other iterators (*select*, *one*, etc.).

OCL allows to define structural properties at the metamodel level in order to validate them on the conforming models. To assess these properties, the DSML end-user can use OCL checkers like for example the Eclipse OCL checker [3].

Considering the SPEM metamodel, Listing 2.1 defines an OCL property which verifies whether the workdefinitions' names are unique on a process. In addition, the OCL property shown in the Listing 2.2 verifies the non-reflexivity of a worksequence.

```
context Process
inv names_uniqueness:
self.workDefinitions
      ->forAll(wd1, wd2|wd1 <> wd2 implies wd1.name <> wd2.name)
```

Listing 2.1 — OCL property verifying the uniqueness of workdefinitions' names

```
context WorkDefinition
inv not_reflexive:
self.predecessor <> self.successor
```

Listing 2.2 — OCL property verifying the non-reflexivity of a worksequence

These properties define the *static* (or *structural*) semantics. It corresponds to defining restrictions on the structure of DSML conforming models.

2.1.2 Concrete syntax of a DSML

A concrete syntax describes a specific representation of the DSML used to display models to end users. It can be either a textual or graphical representation. It eases the use of the abstract syntax concepts and thus the creation of DSML conforming models. Several projects provide tools to implement textual concrete syntaxes for DSLs like Xtext[4], TCS[5], or EMFText[6] or graphical concrete syntaxes like Graphical Modeling Framework[7] (GMF), Sirius[8] and Graphiti[9].

Figure 2.2 shows an example of a process model. It corresponds to a simplified development process composed of four activities, each represented with an ellipse: *Programming*, *Designing*, *Test case writing* and *Documenting*.

The "finishToStart" dependency between *Designing* and *Programming* means that *Programming* can only be started when *Designing* has been finished. *Documenting* and *TestCaseWriting* can start once *Designing* is started (*startToStart*) and *Documenting* cannot finish if *Designing* is not finished (*finishToFinish*).

[3] http://www.eclipse.org/modeling/mdt/?project=ocl
[4] http://www.eclipse.org/Xtext/
[5] http://www.eclipse.org/gmt/tcs/
[6] http://www.emftext.org/
[7] http://eclipse.org/gmf-tooling/
[8] http://www.eclipse.org/sirius/
[9] http://www.eclipse.org/graphiti/

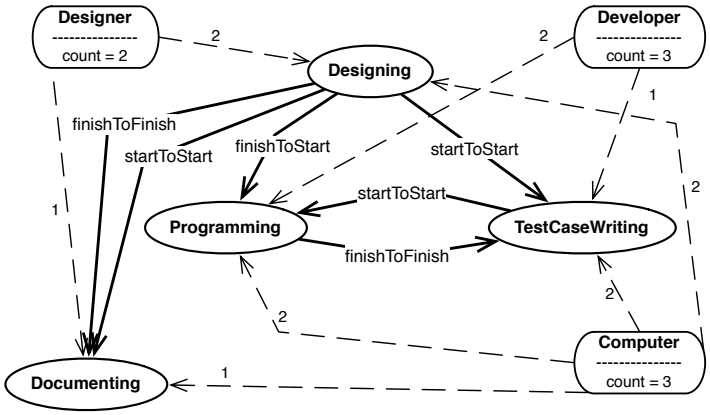

Figure 2.2 — A SPEM development process

```
process Development {
    wd Designing (Designer (2), Computer(2))
    wd Documenting (Designer (1), Computer(1))
    wd Programming (Developer (2), Computer(2))
    wd TestCaseWriting (Developer (1), Computer(2))
    ws finishToFinish from Designing to Documenting
    ws startToStart from Designing to Documenting
    ws finishToStart from Designing to Programming
    ws startToStart from Designing to TestCaseWriting
    ws finishToFinish from Programming to TestCaseWriting
    ws startToStart from TestCaseWriting to Programming
    rs Designer (2)
    rs Developer (3)
    rs Computer (3)
}
```

Listing 2.3 — A textual formalization of the SPEM development process

The dependencies put between *Programming* and *TestCaseWriting* enforce a test driven development: programming can only start when test cases are already started and, obviously, test case writing can only be finished when programming is finished in order to take into account test coverage.

Rounded rectangles represent resources with their amounts (2 *Designers*, 3 *Developers* and 3 *Computers*). Dashed arrows indicate how many occurrences of a resource an activity requires. On Figure 2.2, *Programming* needs two developers and two computers. Resources are allocated when an activity starts and freed when it finishes.

A concrete textual syntax for SPEM can be defined with Xtext. A possible formalization of the SPEM development process can be shown in Listing 2.3.

2.1.3 Behavioral semantics for a DSML

Usually, the behavioral semantics is neglected in the definition of a language. However, as we focus on executable DSML, it is a key feature to define the behavior of a model during

the execution. It extends the static semantics defined on the DSML metamodel which is independent of the execution of a model and can be defined as well-formedness properties expressed with OCL. It is usually implicit as the names of the concepts and relations in the abstract syntax usually carry an intended meaning related to the semantics. However, if we want to correctly understand the signification of a DSML conforming model, it is mandatory to introduce an explicit semantics for the DSML that rigorously defines the meaning of the different DSML constructs. In addition, as we target the critical embedded systems in the POLARSYS project[10], the semantics of a DSML becomes a mandatory element to verify and to validate models defined earlier in the development process. There exists two main approaches to define a behavioral semantics [CRC+06]:

- *operational* semantics (the left side of Figure 2.3): It is expressed in the same technical space used for the definition of the DSML abstract syntax. It describes the execution of a model as a sequence of models expressed in the same language extended to represent the state of the execution at a given step in time. This approach requires the extension of the DSML abstract syntax with the required elements to store the execution information. In the MDE context, to express an operational semantics, two kinds of approaches are proposed. The first one consists in using meta-programming languages like Kermeta or the EMF API for Java to specify imperatively the behavior of different language constructs. This approach extends metaclasses with operations that describe the evolution of a model. The second approach is based on endogenous model transformations expressed on the abstract syntax using a model transformation language like ATL. It allows to declaratively define the behavior as a state transition system based on possible model states.

- *denotational* (*translational*) semantics (the right side of Figure 2.3): It is expressed in a technical space different from the DSML one. The target paradigm should be defined rigorously and adapted to the construction of powerful tools for analysis (like model-checking tools, simulators tools). In the MDE context, it consists in defining an exogenous model transformation that maps the DSML abstract syntax into the formal domain to allow the use of the corresponding tools in this formal domain. We detail this aspect for SPEM in the chapter 3.

2.2 Model verification for DSMLs

As we target safety critical systems, verification and validation (V&V) activities are mandatory. We need to provide high quality tools for DSML end-users in that purpose. Model verification is a critical step in the development process. It consists in assessing the conformance of the designed models to the requirements relying on the DSML semantics.

A lot of activities have been conducted in the last 20 years regarding the integration of formal V&V for DSML (see [BGHM05, BCL+01, RKK08, RL12, Rus11, DMGB09, GdLMD09, RKK08, PIM09, GCKK06]).

[10] `http://polarsys.org/`

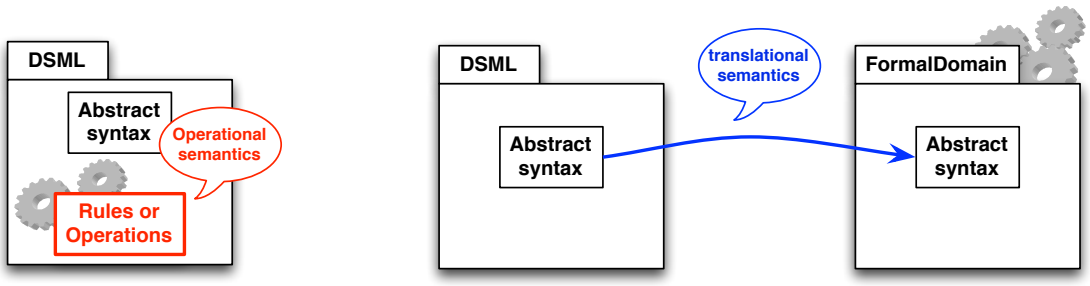

Figure 2.3 — Operational versus translational semantics

We only detail three appropriate activities that are considered as generic solutions. Most of the existing ones are similar to these three.

In [RL12], a formal approach based on the $\mathbb{K}$ semantic framework is proposed to define DSMLs. It helps the DSML designer in formalizing DSML elements using the MDE technical space (metamodels for the DSML abstract syntax, OCL constraints for the static semantics and model transformations for operational semantics). In fact, the approach proposes a textual language to encode the DSML metamodel and another one to define its conforming models. The DSML operational semantics is encoded with KMRL language ($\mathbb{K}$ Model-Rewrite Language) which is a mixed declarative/imperative language for model rewriting. Different operational semantics elements are defined as endogenous model transformation rules. Then, the whole DSML definition in mapped using the Rascal metaprogramming language [KSV09] into the $\mathbb{K}$ semantic framework [RS10] to benefit from $\mathbb{K}$'s execution engine and formal analysis tools. Therefore, an executable semantics is generated. All formal aspects are hidden for the DSML designer who handles only high-level tools proposed in the approach. To perform the verification activity, a KMRL rule is added as an observer which verifies on the executable semantics whether it holds and generates verification results.

In [Rus11], a formal approach is proposed to define and analyse DSMLs. The approach consists in representing DSML metamodels and their conforming models as a Maude specification [CDE$^+$07]. The operational semantics, defined as an endogenous model transformation, is encoded in Maude as rewrite rules. To ask a Maude specification whether a DSML conforming model behaves as expected, the question is formulated as a Maude command.

Defining an operational semantics for a DSML does not show interesting results to guarantee the verification activity because it requires defining a domain-specific model-checker providing an efficient encoding of the state and the execution relation which is not realistic. Most approaches proposed in the literature to deal with the integration of model verification in a metamodeling process are based on defining a translational semantics for the DSML.

In [DMGB09], the authors propose an approach to assist designers in the definition of a behavioral semantics and thus the verification specification for DSMLs using visual languages. In fact, the approach consists in specifying the behavior of the DSML as transformation rules using AGG [Bey92]. AGG is a rule-based visual language supporting an algebraic approach to graph transformation. Each AGG transformation defines the behavior for one

of the state transitions. It is thus sufficient to describe the whole behavior of the DSML. Theses AGG transformations are extended to build a sequence of state changes. These sequence definitions explain the related elements to the execution (the expected transition, the order and the condition of the execution). This information is defined using an activity diagram.

The verification process maps different DSML definition elements (structural and behavioral ones) into an Alloy specification [JSS01]. For instance, the DSML structural elements are transformed into Alloy abstract signatures, the graph transformation rules and their related activity diagrams are mapped into Alloy predicates, the DSML conforming model is transformed into an Alloy concrete signature and finally, a verification task is defined as an Alloy assert. The complete Alloy specification is then checked to find whether it is correct. Otherwise, a counter-example is generated.

3 SPEM as a DSML

Résumé

Dans ce chapitre, nous allons présenter l'étude de cas qui servira à illustrer notre travail. Nous présentons les travaux de vérification faits par Benoît Combemale pendant sa thèse de doctorat. Il propose une approche par métamodélisation pour exprimer la sémantique d'exécution d'un DSML en fournissant une sémantique translationnelle. Cette approche est appliquée à la vérification et à la simulation de modèles de processus exprimés en utilisant le langage SPEM (Software Process Engineering Metamodel).

La sémantique translationnelle de SPEM cible un domaine sémantique formel, qui est les réseaux de Petri temporels (TPN), afin de réutiliser des outils de vérification de modèle (model-checking) existants fournis par la boîte à outils TINA. Nous détaillons les différents travaux effectués pour faciliter l'intégration de la vérification formelle pour un DSML afin d'engendrer automatiquement les propriétés comportementales formelles dans le format approprié pour le model-checker. L'idée consiste à écrire manuellement une transformation de modèle à texte en ATL qui accepte un modèle conforme au DSML et génère les propriétés comportementales formelles. Nous montrons également les résultats de vérification obtenus au niveau formel et obtenus automatiquement grâce au model-checker SELT de la boîte à outils TINA.

En outre, nous présentons un parton de métamodélisation proposé par le même auteur dont le but est d'assister l'expert du DSML à expliciter toutes préoccupations différentes de la sémantique d'exécution d'un DSML et de favoriser la définition d'outils génératifs et ainsi faciliter l'intégration des outils pour de nouveaux DSMLs.

Cette approche présente de nombreux avantages comme l'utilisation des outils puissants mais aussi quelques inconvénients car l'actuel état de l'intégration ne cache pas intégralement les aspects formels. On évaluera cette approche en montrant ce qui est acquis par cette intégration et en détaillant ses inconvénients. Enfin, nous concluons en soulignant l'objectif de notre travail d'étendre l'approche existante afin de nous attaquer aux inconvénients identifiés. Ces objectifs ont été fixés par rapport aux attentes du concepteur, de l'expert et des utilisateurs finaux d'un DSML. Ils consistent principalement à faciliter l'expression des propriétés comportementales au niveau DSML, produire les propriétés formelles depuis ces propriétés comportementales et remonter les résultats de vérification vers le niveau DSML depuis le niveau formel.

IN this chapter, we will introduce the running case-study which aims to illustrate our work. We present the verification task defined by Benoît Combemale in his PhD thesis [Com08]. He proposes a metamodeling approach to express the execution semantics of a DSML thanks to a translational semantics. It is applied to the verification and the simulation of process models expressed with SPEM.

The translational semantics for SPEM targets a formal semantics domain, which is time Petri nets (TPN), in order to reuse existing model-checking tools provided by the TINA toolbox [BRV04]. We relate different works done to ease the integration of formal verification for DSML by automatically generating formal behavioral properties in the appropriate format for the model checker.

In addition, we present a metamodeling pattern proposed by the same author whose purpose is to explicit different concerns of the execution semantics of a DSML and to favor the definition of generative tools and thus ease the integration of tools for new DSMLs.

This approach has many advantages but also some drawbacks which will be detailed. Finally, we conclude by stressing the aim of our work to extend the existing approach in order to tackle identified disadvantages.

3.1 Verification of SPEM models

SPEM has been considered as a running case-study to experiment verification and validation (V&V) activities within the TOPCASED project [Pan07].

Because the TOPCASED toolkit addresses safety critical systems, Verification and Validation (V&V) activities are of primary importance and should be performed as early as possible in the development process and particularly at design time on the various models. The aim is both to reduce the development costs and to provide higher quality systems.

Validation is performed through model animation [CCP+10]: the system designer who is the DSML end-user builds a model using a graphical editor and can execute it according to scenarios. The runtime data produced by these executions is displayed as decorations of the graphical representation of the model or thanks to a dedicated view. Model animation is thus very similar to source level debugging for software. Scenario driven model execution runs through a single path in the set of all possible executions for the model. The use of several scenarios provides a coverage of the various possible executions but this validation is usually not exhaustive.

Verification means checking whether models, which are conforming to the DSML, reflect the DSML requirements. Two kinds of properties are investigated: structural properties and temporal properties [CGCT07]. Once the SPEM structural properties are expressed and verified with an OCL checker, behavioral properties, also named temporal properties, must be addressed. They allow to verify the model during execution to check whether it behaves as expected.

The DSML expert may be interested in general properties not specific to a given process model. For the SPEM example, he may want to check whether a process model may finish or not (P_1 requirement). A process finishes if all its activities finish while respecting constraints

imposed by dependencies and resource allocation. If these properties hold, the DSML end-user may want to get a terminating scenario and use it to pilot the process execution.

The DSML end-user may also want to verify properties that are specific to a particular process model. As an example, considering the process model of Figure 2.2, he might want to know if in all cases *Documenting* is finished before *Designing* is finished (P2).

To assess these properties, model execution is required. The adopted approach in the literature consists in defining a translational semantics into a well-formed mathematical technical space in order to reuse existing powerful tools like model-checkers, simulators, etc.

3.1.1 Time Petri nets, SE-LTL and Tina toolbox

In this study, the technical space of time Petri nets is chosen to formally express the SPEM semantics. We have also chosen to express our temporal properties as SE-LTL formulae (State/Event Linear Temporal Logic) over the time Petri net associated to a SPEM model. Then, we manipulate them within the TINA toolkit.

Time Petri nets (or TPN) [MF76] is one of the most widely used model for the specification and verification of real-time systems. TPNs are Petri nets in which a non-negative real interval Is(t), with rational end-points, is associated with each transition t of the net [MF76]. When a transition is enabled, a clock starts and the transition can only be fired when the clock is in the transition time interval. This ensures decidable verification for bounded Petri nets which is not the case for temporal automaton.

Definition 5. *A* TPN *is a tuple* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$*, in which* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ *is a Petri net, and* $I_s : T \rightarrow \mathbf{I}^+$ *is the static interval function.*

P is the set of places, T is the set of transitions, $\mathbf{Pre}$, $\mathbf{Post} : T \rightarrow P \rightarrow \mathbf{N}^+$ *are the precondition and postcondition functions,* $m_0 : P \rightarrow \mathbf{N}^+$ *is the initial marking.* $\mathbf{I}^+$ *is the set of nonempty real intervals with nonnegative rational end-points. The right one might be infinite* ∞*.*
Let $\mathbf{R}^+$ *be the set of nonnegative reals. For* $i \in \mathbf{I}^+$, $\downarrow i$ *denotes its left end-point, and* $\uparrow i$ *its right end-point (if i bounded) or* ∞*. For any* $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ *denotes the interval* $\{ x - \theta \mid x \in i \wedge x \geq \theta \}$*.*

States and the temporal state transition relation $\xrightarrow{t@\theta}$ are defined as follow:

Definition 6. *A state of a* TPN *is a pair s= (m, I) in which m is a marking and I is a function called the interval function. Function* $I : T \rightarrow \mathbf{I}^+$ *associates a temporal interval with every transition enabled at m.*

We write $(m, I) \xrightarrow{t@\theta} (m', I')$ *if* $\theta \in \mathbf{R}^+$ *and:*

1. $m \geq \mathbf{Pre}(t) \wedge \theta \geq \downarrow I(t) \wedge (\forall\, k \in T)(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
2. $m = m' - \mathbf{Pre}(t) + \mathbf{Post}(t)$
3. $(\forall\, k \in T)(m' \geq \mathbf{Pre}(k) \Rightarrow$

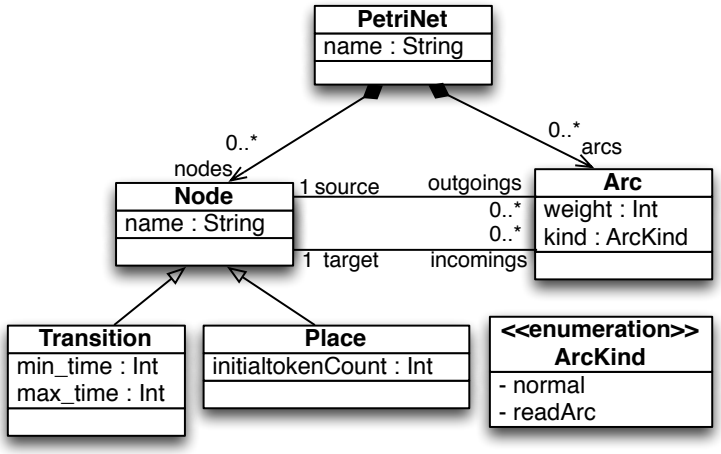

Figure 3.1 — Time Petri net metamodel

$$I'(k) = \begin{array}{l} \textbf{\textit{if}}\ k \neq t \wedge m - \textbf{\textit{Pre}}(t) \geq \textbf{\textit{Post}}(t) \\ \textbf{\textit{then}}\ I(k) \dot{-} \theta \\ \textbf{\textit{else}}\ I_s(k)) \end{array}$$

TPN metamodel The TPN metamodel is shown in Figure 3.1. It is composed of nodes (*Node*) that denote places (*Place*) or transitions (*Transition*). Nodes are linked together by arcs (*Arc*). Arcs can be normal ones or read-arcs (*ArcKind*). The attribute *initialtokenCount* specifies the number of tokens consumed in the source node or produced in the target one (in case of a read-arc, it is only used to check whether the source place contains at least the specified number of tokens). Finally, a time interval can be expressed on transitions.

Model-Checking For this study, we use State/Event LTL (SE-LTL) [CCO+04], a linear time temporal logic supporting both state and transition properties. The modeling framework consists of labeled Kripke structures (the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions.

Formulae Φ of State/Event LTL are defined according to the following minimal grammar:

$\Phi ::= p \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc \Phi \mid \Box \Phi \mid \Diamond \Phi \mid \Phi \, U \, \Phi$

Let's show some SE-LTL properties:

	(For all paths)
P	P holds at the beginning of the path,
$\bigcirc$ P	P holds at the next step,
$\Box$ P	P globally holds in all steps,
$\Diamond$ P	P holds in a future step,
P U Q	P holds until a step is reached where Q holds

where a *path*, named also execution, is a possible infinite sequence alternating states and transitions.

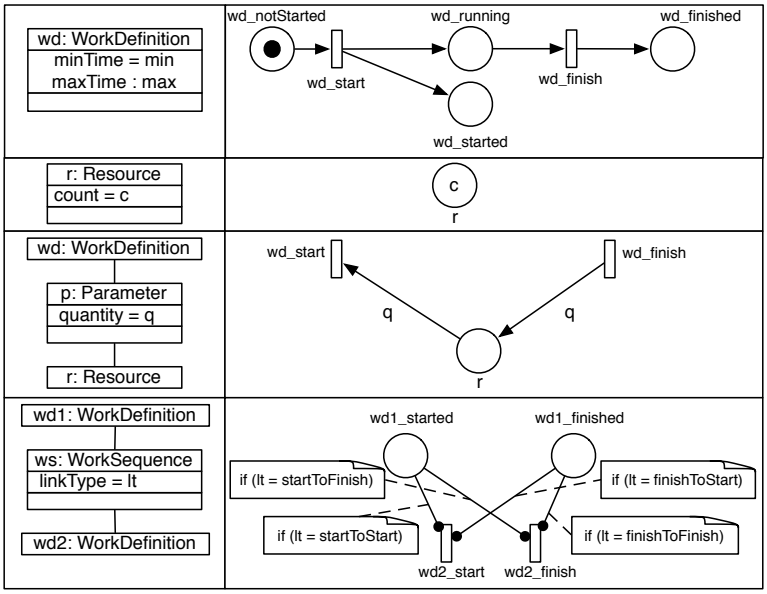

Figure 3.2 — The translational semantics of SPEM into TPN

Tina Toolbox for Time Petri Nets Verification [BRV04] is a toolbox for edition and analysis of Petri Nets and Time Petri nets, developed by the OLC group of LAAS/CNRS. Among its available tools, we rely in this work on:

- nd (NetDraw): graphical or textual editor for (Time) Petri Nets, including a simulator.
- TINA: this tool, with the same name of the toolbox, allows the construction of reachability graphs and Kripke transitions systems - useful for the verification by model-checking - from Petri Nets, for example.
- selt: allows the user to provide SE-LTL formulas and verify if the Kripke transitions system - generated by TINA- satisfies them. When a property is not verified, the tool returns a counterexample - which can be simulated by TINA.

3.1.2 Translational semantics of SPEM into Petri nets

Several translational semantics for SPEM can be defined according to the level of details in the execution that we want to model and the kind of properties we want to assess. Thus, Benoît Combemale advocates in [CCG+07] that defining the translational semantics should be *property-driven* to favor the definition of a *minimal* semantics, that will allow to answer to the questions the user may ask about his models. This approach has been developed and experimented by Ning Ge in her PhD [Ge14]. Our work targets processes, methods and tools to ease its implementation.

Here is some rationale behind the translational semantics shown in Figure 3.2. A *WorkDefinition* is translated into four places characterizing its state (*notStarted*, *started*, *running* and *finished*) linked by two transitions. These transitions model the actions that we want

to observe on a workdefinition: one can *start* a workdefinition and then *finish* it. A workdefinition is considered *started* if it is either running or finished. This is recorded by the place named *started*.

A *WorkSequence* becomes a *read-arc*[1] from one place of the source workdefinition (either *started* or *finished*) to a transition of the target workdefinition (either *start* or *finish*) according to the kind of *WorkSequence* (*linkKind* attribute). A resource becomes a place whose initial marking (*initialtokenCount*) corresponds to its *count*. Each *Parameter* element is translated into two arcs, the first one to take resources when the concerned workdefinition starts and the second one to release them when the workdefinition finishes.

3.1.3 Expressing and generating formal properties

Based on the translational semantics of SPEM into TPN, temporal properties on a SPEM model can be automatically generated into formal ones expressed on TPN model.

Taking the SPEM model described in the Figure 2.2, it is mandatory to generate a SE-LTL file holding the formulas to be verified . It is shown in Listing 3.1 and contains three key elements.

First, a *finished_process* operator is defined. It formalizes the P_1 requirement defined previously. In TPN, a SPEM process is finished if and only if all its workdefinitions are finished. According to the defined translational semantics, it can be shown as the conjunction of *finished* states of its workdefinitions. (one token in the corresponding finished place of the workdefinition).

Then, a SE-LTL property states that a process can never be finished. If it is satisfied, it means that the process cannot be finished, and if it is not satisfied, the process can finish and the model checker would exhibit a counter example that corresponds to a scenario that finishes the process and thus all its activities.

Finally, a second SE-LTL property indicates whether a process can finish. If it does not hold, a counter-example that explains the deadlock is generated.

```
op finished_process = T /\ Designing_finished /\ Documenting_finished /\
   Programming_finished /\ TestCaseWriting_finished;

[] ( - ( finished_process ) );
<> finished_process;
```

Listing 3.1 — The expected LTL properties

To obtain these properties, previous works [CCBV07, Com08] consist in defining an ATL query which generates SE-LTL formulas on the TPN model from a SPEM model. It is shown in Listing 3.2. This query generates a SE-LTL file named *finished.ltl* (line 1). It combines at the same time: SE-LTL syntax elements as operators (*always*, *eventually*, etc.), OCL expressions which query a SPEM model and some elements related to the defined translational semantics. All these elements ease the automatic generation of SE-LTL properties. Running this

[1] A read-arc only checks that there is enough tokens in the input place but those tokens are not withdrawn when the transition is fired.

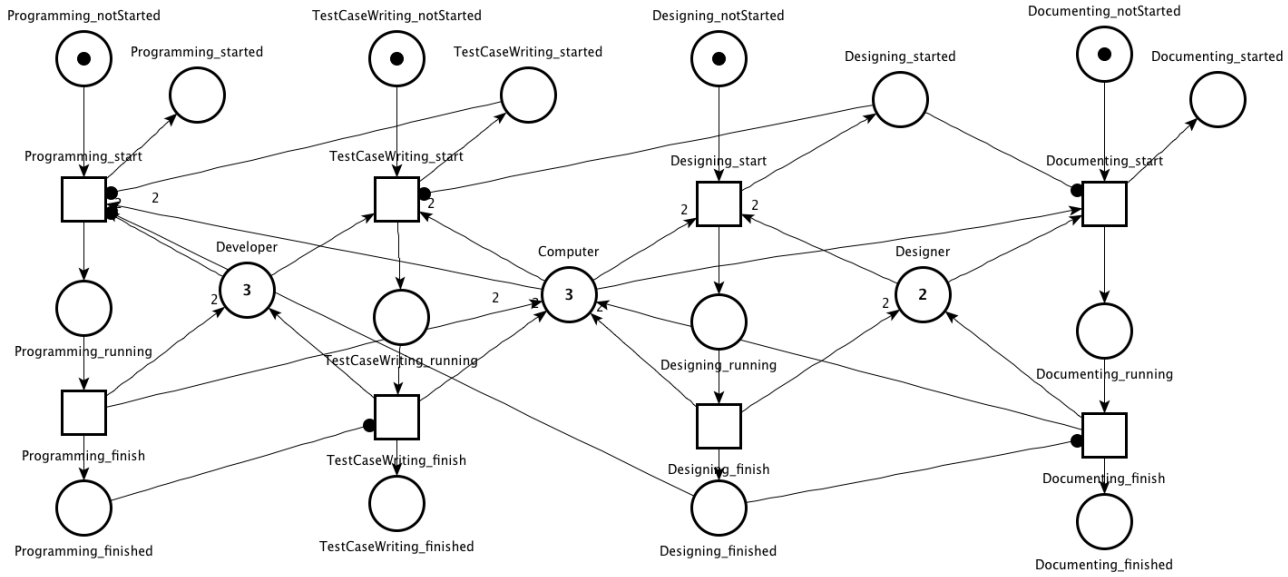

Figure 3.3 — A graphical TPN model generated by performing the translational semantics on the SPEM model shown in Figure 2.2

query with the SPEM model shown in Figure 2.2 allows to generate LTL properties shown in Listing 3.1.

```
query finished=thisModule.generateLTL().writeTo('/SPEM/finished.ltl');

helper def : generateLTL() :String=
-- the finished operator
 'op finished_process = T'.concat( SPEM!WorkDefinition.allInstances()
 ->iterate(wd; acc : String='' |
     acc.concat('/\ ' + wd.name + '_finished')) + ';\n\n'

-- properties :
-- willNeverFinish
     + '[] ( - ( finished_process ) );\n'

-- willEventuallyFinish
     + '<> finished_process;\n\n');
```

Listing 3.2 — An ATL query to generate SE-LTL properties on the TPN model

3.1.4 Performing the formal verification

Once the different steps mentioned above are performed, we can now proceed to the formal verification using the TINA toolbox and more precisely the SELT model checker. Listing 3.3 shows the results produced by the SELT model-checker when verifying the properties of Listing 3.1 on the TPN (Figure 3.3) corresponding to the process model of Figure 2.2.

Because the first property evaluates to true, we can conclude that the corresponding

SPEM process cannot be finished.

Furthermore, the second property which queries whether a process may finish does not hold. the SELT model checker builds a counter example explaining the deadlock. The counter-example contains a set of traces showing the evolution of the TPN during the formal verification (state keyword). A *trace* is a finite sequence of the system states capturing the system during execution. Each trace shows the actual marking of different states. Between each couples of successive traces, it is shown a TPN event corresponding to the fired TPN transitions (lines 8, 11, 14, 17 and 20). The last event (line 24) is an internal event in the SELT model-checker which corresponds to the deadlock.

The counter-example indicates that *Designing* workdefinition starts (*Designing_start*) and finishes (*Designing_finish*). Then, *Documenting* workdefinition starts (*Documenting_start*) and finishes (*Documenting_finish*). *TestCaseWriting* workdefinition starts (*TestCaseWriting_start*) but does not finish. Finally, there is a deadlock in the counter example (*L.deadlock*). *TestCaseWriting* cannot be finished because it requires *Programming* to be finished. However, *Programming* cannot be started because a *Computer* is missing.

```
1  operator finished : prop
2  0.000s
3  TRUE
4  0.001s
5  FALSE
6  state 0: Computer*3 Designer*2 Designing_notStarted Developer*3 Documenting_notStarted
7           Programming_notStarted TestCaseWriting_notStarted
8    -Designing_start->
9    state 1: Computer Designing_running Designing_started Developer*3 Documenting_notStarted
10            Programming_notStarted TestCaseWriting_notStarted
11   -Designing_finish->
12   state 2: Computer*3 Designer*2 Designing_finished Designing_started Developer*3
13            Documenting_notStarted Programming_notStarted TestCaseWriting_notStarted
14   -Documenting_start->
15   state 3: Computer*2 Designer Designing_finished Designing_started Developer*3 Documenting_running
16            Documenting_started Programming_notStarted TestCaseWriting_notStarted
17   -Documenting_finish->
18   state 4: Computer*3 Designer*2 Designing_finished Designing_started Developer*3 Documenting_finished
19            Documenting_started Programming_notStarted TestCaseWriting_notStarted
20   -TestCaseWriting_start->
21 * [accepting] state 5: L.dead Computer Designer*2 Designing_finished Designing_started Developer*2
22                          Documenting_finished Documenting_started Programming_notStarted
23                          TestCaseWriting_running TestCaseWriting_started
24   -L.deadlock->
25   state 5: L.dead Computer Designer*2 Designing_finished Designing_started Developer*2
26            Documenting_finished Documenting_started Programming_notStarted TestCaseWriting_running
27            TestCaseWriting_started
28 0.001s
```

Listing 3.3 — A TPN counter example explaining the deadlock

If we add a computer and run again the formal verification, the first property fails and the second property holds. Analyzing the counter example, the SELT model checker generates a terminating scenario that finishes the process and thus all its activities. The scenario in Listing 3.4 shows a first part already generated in the previous counter-example (Listing 3.3), before the deadlock occurs, extended with the start of *Programming* (*Programming_start*) because there is now enough computer, then*Programming* finishes (*Programming_finish*) and, finally, *TestCaseWriting* can finish (*TestCaseWriting_finish*).

```
operator finished : prop
0.000s
FALSE
  state 0: Computer*4 Designer*2 Designing_notStarted Developer*3 Documenting_notStarted
           Programming_notStarted TestCaseWriting_notStarted
  -Designing_start->
  state 1: Computer*2 Designing_running Designing_started Developer*3 Documenting_notStarted
           Programming_notStarted TestCaseWriting_notStarted
  -Designing_finish->
  state 2: Computer*4 Designer*2 Designing_finished Designing_started Developer*3
       Documenting_notStarted Programming_notStarted TestCaseWriting_notStarted
  -Documenting_start->
  state 3: Computer*3 Designer Designing_finished Designing_started Developer*3 Documenting_running
       Documenting_started Programming_notStarted TestCaseWriting_notStarted
  -Documenting_finish->
  state 4: Computer*4 Designer*2 Designing_finished Designing_started Developer*3 Documenting_finished
       Documenting_started Programming_notStarted TestCaseWriting_notStarted
  -TestCaseWriting_start->
  state 5: Computer*2 Designer*2 Designing_finished Designing_started Developer*2 Documenting_finished
       Documenting_started Programming_notStarted TestCaseWriting_running TestCaseWriting_started
  -Programming_start->
  state 6: Designer*2 Designing_finished Designing_started Documenting_finished Documenting_started
       Programming_running Programming_started TestCaseWriting_running TestCaseWriting_started
  -Programming_finish->
  state 7: Computer*2 Designer*2 Designing_finished Designing_started Developer*2 Documenting_finished
       Documenting_started Programming_finished Programming_started TestCaseWriting_running
       TestCaseWriting_started
  -TestCaseWriting_finish->
  state 8: L.dead Computer*4 Designer*2 Designing_finished Designing_started Developer*3
       Documenting_finished Documenting_started Programming_finished Programming_started
       TestCaseWriting_finished TestCaseWriting_started
  -L.deadlock->
  state 9: L.dead Computer*4 Designer*2 Designing_finished Designing_started Developer*3
       Documenting_finished Documenting_started Programming_finished Programming_started
       TestCaseWriting_finished TestCaseWriting_started
  [accepting all]
0.002s
TRUE
0.001s
```

Listing 3.4 — A TPN terminating scenario

3.1.5 Implementation of the approach

As shown in Figure 3.4, the proposed approach contains three steps. The first step consists of a translational semantics implemented using the ATL transformation language. An ATL module (SPEM2TPN.atl) describes the transformation from a SPEM model (myProcess.spem as shown in Figure 2.2) to a TPN model (myProcess.tpn as shown in Figure 3.3) and, an ATL query (TPN2Tina.atl) generates the textual model (myProcess.net) used by the TINA tools from a TPN model. Obviously, this ATL query is independent of the translational semantics.

Based on the defined translational semantics, the second step consists in automatically generating formal properties (properties.ltl as shown in Listing 3.1) thanks to an ATL query (SPEM2LTL.atl as shown in Lisitng 3.2)

Once both steps are performed, the SELT model checker of the TINA toolbox can be used to generate verification results as a counter-example for properties that do not hold (re-

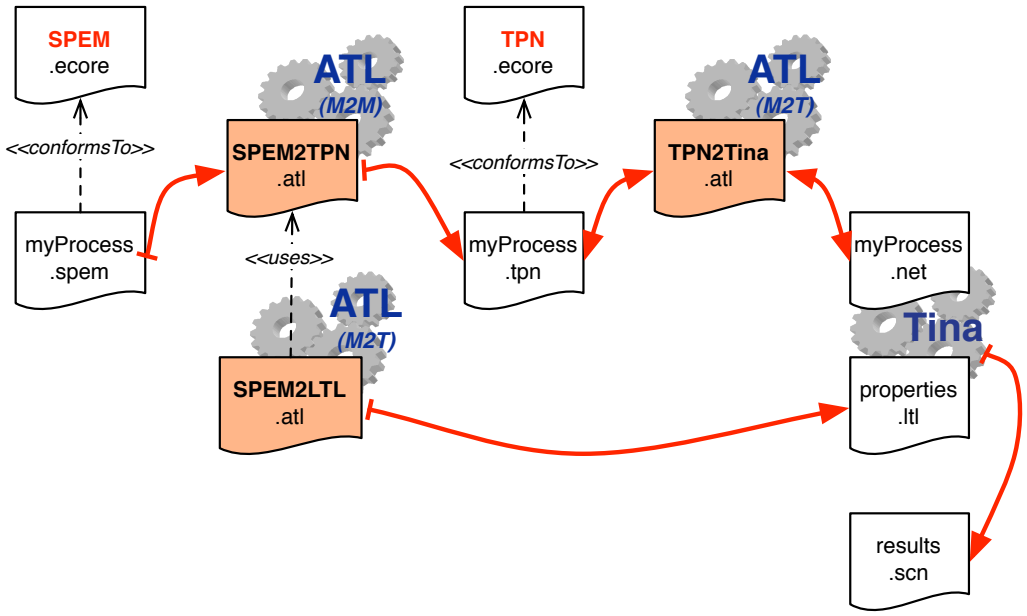

Figure 3.4 — An approach to verify behavioral properties on a process model conforming to SPEM using TPN

sults.scn as shown in Listing 3.3 and Listing 3.4).

3.2 Towards the definition of an eXecutable DSML (*xDSML*)

As shown in the last subsection, model executability is a key concern in MDE to introduce behavioral V&V in the development process. It illustrates the evolution of the model over time. The definition of the execution semantics for DSMLs requires extending the DSML metamodel with the necessary elements to capture the additional dynamic information from the execution. To help in the extension of a new DSML with runtime information, we choose to refer to the *Executable DSML pattern* proposed in [CCP12].

3.2.1 The *Executable DSML pattern*

The *Executable DSML pattern* was proposed as a general and reusable approach to assist the DSML expert in the definition of an execution semantics for a DSML. It allows to make explicit the various concerns from the execution of DSMLs.

It targets the automation of the implementation of DSML tools for V&V. It has been proposed to ease the development of V&V tools in the TOPCASED project. It eases the model V&V by providing graphical model animation for DSMLs [CCP+10]. In the following, we detail different elements of the *Executable DSML pattern* illustrated with the SPEM metamodel.

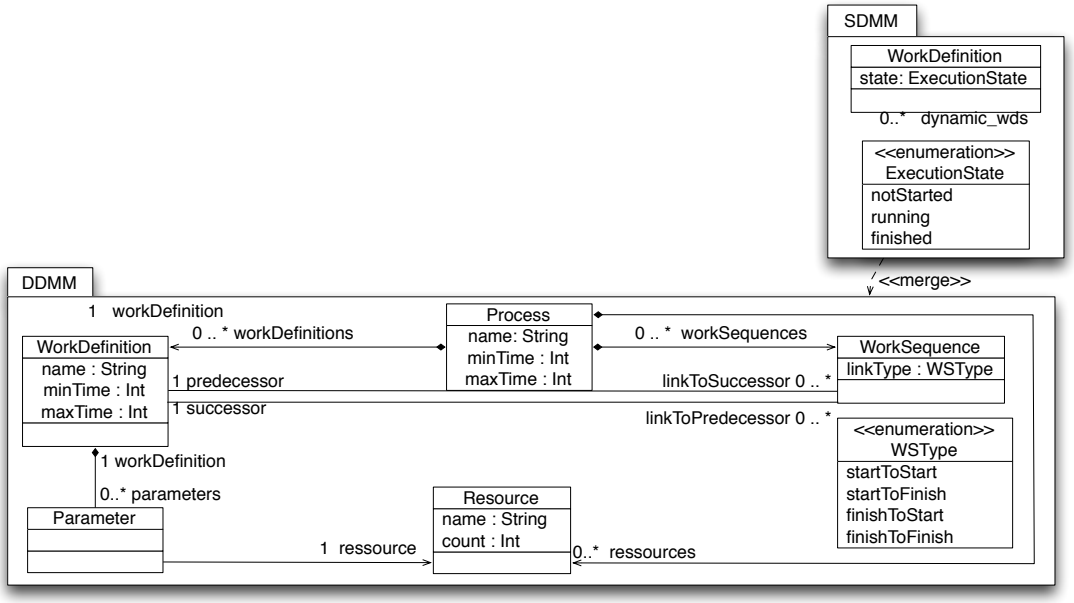

Figure 3.5 **—** Introducing the SDMM extension on the SPEM metamodel

3.2.1.1 Domain Definition MetaModel (DDMM)

The *Domain Definition MetaModel* (DDMM) is the usual metamodel. It provides the key concepts of the considered domain and their relationships. It is the metamodel defined with metamodeling language like ECORE, KM3, etc. This metamodel can be extended with static constraints to assess structural properties.

Usually, this metamodel lacks information related to the execution of model. For example, the state of a workdefinition or the number of available resources are not represented.

For SPEM, the original metamodel shown in Figure 2.1 is the DDMM. It shows only static information related to the structure of a modeling language (meta-classes, relationships, etc.).

3.2.1.2 State Definition MetaModel (SDMM)

The *State Definition MetaModel* (SDMM) defines the runtime information i.e., information that changes during the model execution. It is related to the DDMM by the «merge» predefined package operator [OMG06].

During the execution of a model, additional data may be generated. This information shows the state of a model during its execution. To record these data, a possible extension to the DDMM can be defined.

Figure 3.5 shows the first extension, the SDMM which defines the runtime information, that is the data that model the state of the model at runtime and that are not part of the

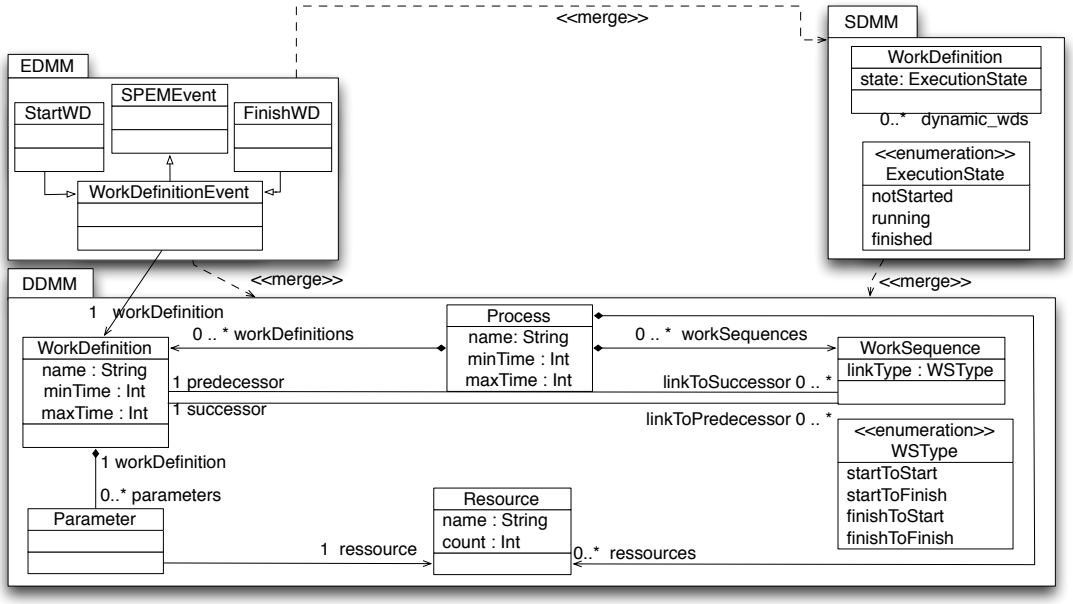

Figure 3.6 — Defining different events can be captured on the SPEM metamodel

DDMM.

For SPEM, the SDMM includes the achievement state of a workdefinition which is either *not started*, *running* or *finished*.

3.2.1.3 Event Definition MetaModel (EDMM)

The *Event Definition MetaModel* (EDMM) implements the concrete stimuli of the DSML that makes a conforming model evolves. Concrete EDMM events are in relation with events related to the defined formal semantics. These events allow to show how a DSML conforming model evolves.

Figure 3.6 introduces a second extension, EDMM, which describes stimuli that make the model evolve. They are modeled as events. *Start a WorkDefinition* or *Finish a WorkDefinition* are examples of such SPEM events.

3.2.1.4 Trace Management MetaModel (TM3)

The *Trace Management MetaModel* (TM3) defines elements to model a scenario (either an input scenario or the trace of a particular execution) as a sequence of event occurrences. It is given in Figure 3.7. TM3 is not specific to one particular DSML as it only relies on the abstract Event concept. It allows to represent a scenario as a succession of domain-specific events that are already defined in the EDMM.

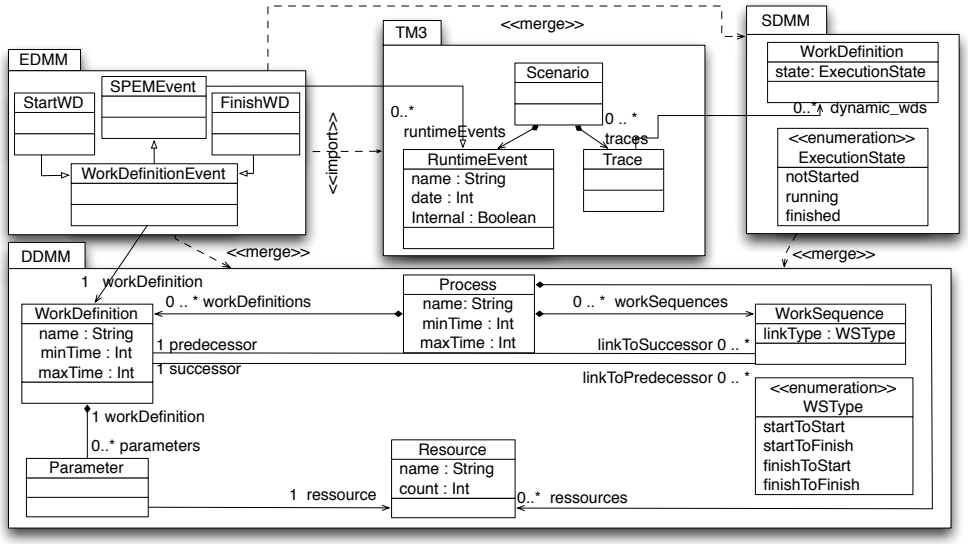

Figure 3.7 — The *Executable DSML pattern* applied into the SPEM metamodel

3.2.2 Application of the Executable DSML pattern to TPN

In order to explicit the execution of a TPN model, it is also required to extend the semantic domain of TPN. Its metamodel is composed of several parts (Figure 3.8). The DDMM describes the abstract syntax of TPN shown previously in Figure 3.1.

The SDMM (State Definition Metamodel) defines an attribute to capture the current count of tokens in a place.

Finally, the EDMM defines only one event *FireTransitionEvent* and, obviously, the TM3 is the same as the one presented for XSPEM, as it is DSML-independent.

3.3 The evaluation of the approach

The adopted approach in this work consists in verifying the behavior of DSML models thanks to model checking techniques. It was performed using model transformation techniques to define a translational semantics from the DSML (SPEM) to a formal language (TPN) in order to benefit of the TINA toolbox capabilities.

Combining user-friendly and automation from the MDE with rigorous and powerful formal method technique allows to capitalize on advantages of both approaches. Nevertheless, both approaches have drawbacks that have to be tackle down in order to gain maximum benefits from their coupling. These are detailed here after.

In [GRS10], a study was developed to show capabilities by combining formal methods and MDE techniques to ease model-driven system design and analysis. In addition, different disadvantages of this combination were detected. Based on the presented approach of

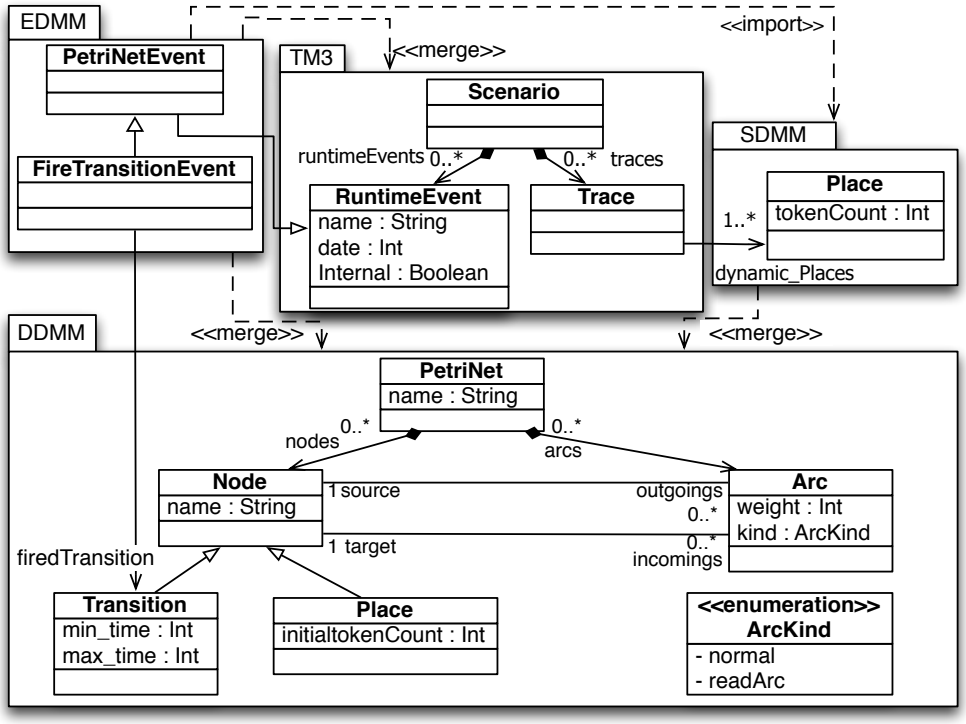

Figure 3.8 — The application of the *Executable DSML pattern* into the TPN metamodel

SPEM verification models, we show different resolved disadvantages. Next, we detail what disadvantages still needs to be solved for the DSML end-user and also for the DSML expert and designer.

3.3.1 Resolved MDE disadvantages

MDE technologies promote models as first-class artifacts. Metamodeling is a key feature of the MDE paradigm. It already provides means to define the abstract syntax of DSMLs as metamodels, complete them by static properties, and textual and graphical concrete syntaxes. However, the semantics definition of these languages appears as a crucial challenge.

Therefore, **lack of semantics** can be considered as a major disadvantage of the MDE approach. In the presented work of SPEM models verification using TPN, this disadvantage was solved by providing a translational semantics into a rigorous mathematical technical space which is TPN. An operational semantics could also have been defined [CCP+10, CHJ+12]. However, the validity of different proposed semantics is still a challenge because it lacks a reference semantic for each DSML that can be used to check whether different defined semantics respect it.

A second disadvantage can appear for the MDE approach, which is **the unfitness for model analysis**. Due to the lack of semantics for metamodel-based languages, performing model analysis is not possible. It can be considered as a consequence of the first cited disadvantage whose resolution can be a way to resolve the second one. Thus, this disadvantage

has disappeared after connecting the TINA toolbox to the DSML through the translational semantics and then verifying the generated TPN. If the formal formulas (properties.ltl) don't hold, SELT model-checker exhibits a counter example (results.scn): a specific execution of the model that leads to a state where the property is not satisfied.

3.3.2 Unresolved formal methods disadvantages

The application of formal methods (especially, model checking and static analysis techniques) for the verification of safety critical embedded systems has produced very good results due to their rigorous mathematical foundations and raised the interest of system designers up to the application of these technologies in real size projects. They allow to detect errors and bugs at earlier phases of the development process.

However, these methods usually rely on complex specific verification-oriented formal languages that most system designers do not master. Their **mathematical nature** makes defining translational semantics a more difficult task for the developer due to the semantic gap between these formal languages and metamodeling languages (general-purpose languages or domain-specific languages).

In the presented work, this disadvantage does not appear clearly due to the simplicity of the subset of the SPEM language chosen for the case study. However, once we decide to define a translational semantics for more complex metamodeling languages where it is expected to show data exchange or time constraints, it becomes a barrier for DSML designers to understand and translate high-level constructs to low-level formal languages.

A second disadvantage is the **lack of tools**: formal methods most of the time don't offer easy-to-use tools to assist a developer during the development process. Also, most of combination approaches do not target the integration of formal methods in a seamless manner which hides all formal aspects for the system designer.

The approach proposed in Combemale PhD thesis does not offer a hidden use of formal methods for DSML end-users. In addition, for the DSML expert and designer, it is not easy (1) to express the behavioral properties on the formal side and (2) to define the ATL query to generate behavioral properties. In addition, (3) the DSML end-user cannot define specific behavioral properties for its models. For him, editing the ATL query is a hard task because he generally does not have strong background on formal verification technologies.

The last emerged disadvantage concerns the **lack of integration**: it is required to integrate formal methods and their associated tools to ease their use and, thus, to benefit from the formal verification results. The most relevant drawback in the presented approach concerns verification results: the DSML end-user cannot interpret the generated verification results by the SELT model checker because he does not have (and should not acquire) a solid knowledge on formal languages and associated tools.

3.4 Goals

This PhD thesis will illustrate our proposals to overcome these disadvantages for the use of formal methods to verify DSML models. We aim, on the one hand, to provide a seamless approach for the DSML end-user to integrate formal verification in a MDE process, and on the other hand, to target a DSML-independent approach, for the DSML expert and designer, that favors the definition of generative tools and thus eases the integration of tools for new DSMLs. We will classify the goals relative to the appropriate actors.

3.4.1 DSML end-user expectations

After designing a DSML conforming model, the DSML end-user usually wants to assess that the model has the expected properties. To ensure a good performance for this task, it is mandatory to provide him with a kind of DSML verification framework to assist him during this work.

First, (1) this framework should provide a toolchain which verifies automatically behavioral requirements formalized previously by the DSML expert and designer. This step shows whether the model behaves as expected.

Second, (2) it should allow the DSML end-user to formalize another kind of behavioral properties, named specific properties, which are specific to a given model.

Finally, (3) the DSML verification framework should provide the DSML end-user with verification results. These results should be relative to the already defined model in order to ease their understanding and, thus, to allow their corrections.

To summarize, the expected framework should only show to the DSML end-user different elements which interact with his model without revealing the formal aspects. These ones should be hidden to provide a seamless framework.

3.4.2 DSML expert and designer expectations

The DSML expert and designer handle the generation of the DSML verification framework. This work contains three key tasks: 1) defining the various requirements that should be verified, 2) defining the translational semantics from the DSML abstract syntax to a formal language which should ease the express of the behavioral requirements, 3) expressing generic behavioral properties and easing the definition of specific behavioral properties by the DSML end-user, and finally, 4) managing the feedbacks of verification results into the DSML level.

As we aim to facilitate the development of CASE tools for new DSML and, thus, we focus on generic and generative approaches advocated by MDE, we decide to assist the DSML expert and designer during his work by providing the necessary tooling and methodology.

It is mandatory to choose the appropriate target tools in order to map the DSML abstract syntax elements. We try to help the DSML designer to reduce the semantic gap between the DSML and low-level formal methods by introducing an intermediate formal level.

Next, we aim to resolve the current lack of tools by providing a high-level tooling to ease the definition of behavioral properties (generic and specific ones) and how to manage theme to generate formal properties automatically.

Finally, in order to ease the integration of tools for new DSMLs and instead of defining an *ad-hoc* way to manage the feedback of verification results for each DSML, we propose to provide for the DSML designer some techniques and tools to support him during this critical step.

Part

Contribution

4 Expressing and verifying behavioral properties

Résumé

Dans la première partie de cette thèse, nous avons présenté les notions de base autour la métamodélisation et la vérification de modèle. Ensuite, nous avons montré notre étude de cas qui permet de souligner les différents avantages de l'intégration de la vérification formelle dans un processus de métamodélisation. De plus, nous avons identifié les différents éléments manquants pour obtenir une approche transparente pour vérifier les propriétés comportementales génériques (liées au métamodèle du DSML) sur des modèles conformes au DSML.

L'un de ces éléments manquants est la spécification et la transformation générique de propriétés comportementales sur les DSMLs. En fait, il est indispensable de fournir pour l'expert et le concepteur d'un DSML les outils nécessaires pour interroger un modèle, formaliser les propriétés comportementales et ensuite les traduire pour pouvoir les vérifier en utilisant des outils de model-checking. En outre, ces outils devraient également faciliter au utilisateur final du DSML la spécification de ses propriétés comportementales spécifiques à ses modèles sur lesquels elles seront évaluées.

Ainsi, il est obligatoire de fournir un langage approprié pour exprimer des propriétés comportementales et de compléter le patron de métamodélisation appliqué au DSML pour identifier les requêtes que l'utilisateur souhaitera poser sur le modèle en cours d'exécution. Nous présentons dans ce chapitre notre première contribution qui vise faciliter l'expression des propriétés comportementales au niveau du DSML et leur traduction vers le niveau formel.

Nous détaillons les différentes étapes proposées pour automatiser ce travail. Tout d'abord, en se basant sur une proposition de Paul Ziemann et Martin Gogolla, nous mettons en œuvre une extension temporelle de OCL, appelée TOCL, qui permet de définir les propriétés comportementales. On s'intéresse principalement aux opérateurs temporels orientés futur (*always*, *eventually*, *next*, etc.). En outre, nous étendons le patron de métamodélisation en explicitant la définition des requêtes liées à l'exécution d'un modèle conforme à un DSML sous la forme du QDMM (Query Definition MetaModel). Ces requêtes peuvent être exprimées en utilisant l'éditeur de propriétés TOCL.

Nous décrivons ensuite comment traduire automatiquement ces propriétés formelles

pour qu'elles puissent être vérifiées par le model-checker. Cette traduction s'appuie sur une abstraction de la sémantique de traduction utilisée.

Nous concluons ce chapitre par une comparaison avec les travaux relatifs.

IN the first part of this thesis, we have presented basic notions around metamodeling and model verification. Then, we have shown our running case-study which allows to point out different advantages of integrating formal verification on metamodeling tasks. In addition, we have identified different missing elements to obtain a seamless approach to verify generic (related to the DSML metamodel) behavioral properties on DSML conforming models.

One of these missing elements is the specification and the verification of behavioral properties on DSMLs. In fact, it is mandatory to provide for the DSML expert and the DSML designer the required tools to query a model, to formalize the behavioral properties and then to translate them in order to be verified later using model-checking tools. In addition, these tools should also ease for the DSML end-user the specification of his model-specific behavioral properties to assess them on DSML conforming models [ZCP12].

Thus, it is mandatory to provide a suitable language to express behavioral properties based on extensions of the DSML metamodel because the current proposed implementation of the *Executable DSML pattern* in Combemale PhD does not favor the definition of this kind of information. We present in this chapter our first contribution which targets easing the expression of behavioral properties at the DSML level and their translation to the formal level.

We detail the different steps proposed to automate this task. First, based on a proposal by Paul Ziemann and Martin Gogolla in [ZG02], we implement a temporal extension of the Object Constraint Language (OCL) [OMG03b] which allows to define behavioral properties. In addition, another kind of information must be modeled in the *Executable DSML pattern*. It provides the DSML queries related to the runtime of a DSML conforming model. We introduce this additional extension for the *Executable DSML pattern* named the *Query Definition MetaModel (*QDMM*)*. These DSML queries can be expressed using the proposed temporal extension of OCL (TOCL) editor.

Then, it is mandatory to verify these formalized properties. We explain the proposed translation to automatically generate the corresponding formal properties of the model checker.

We conclude this chapter by a comparison with related work.

4.1 The expression of behavioral properties

4.1.1 The temporal extension of OCL

OCL is used to define structural properties on models. Initially, It was considered as a constraint language to overcome the limitations of the graphical notation of UML but quickly it became a key feature of any MDE technique [CG12]. OCL a formal language to express side-effect-free constraints. It provides *navigation operators* to access the content of models, *collection operations* and *quantifiers (universal/existential)* to define first order logic statements.

Nowadays, OCL is used as a language component to implement several MDE techniques like model transformations (as ATL, QVT [OMG11a], etc.), well-formedness rules to

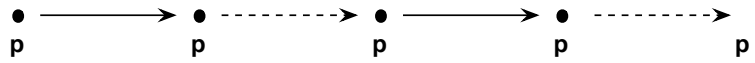

Figure 4.1 — The always temporal operator

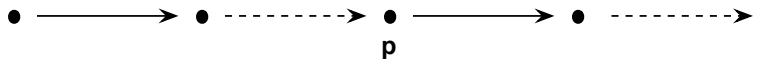

Figure 4.2 — The eventually temporal operator

define new domain specific languages (as OCLinEcore[1]) and code generation templates (as ACCELEO[2]).

It is now a widely known language and a few temporal extensions of OCL have been proposed in order to specify event-based behavioral properties whereas OCL only targets structural properties. We have chosen to rely on TOCL (Temporal OCL) and especially on the proposal from [ZG02] as the syntax of this extension is quite natural for OCL users. It introduces usual future-oriented temporal operators such as always, sometimes, next, existsNext as well as their past-oriented duals. In the following, we list different adopted temporal operators. We illustrate them with diagrams showing the evolution of the execution during time (an execution path) starting from the current state.

4.1.1.1 always operator

The *always* operator, named also *Globally*, is a unary temporal operator in SE-LTL. It is symbolised by $\Box$ or **G**. As shown in Figure 4.1, $\Box$ **p** means that **p** has to hold on the entire subsequent path.

4.1.1.2 eventually operator

The *eventually* operator, named also *Finally*, is a unary temporal operator in SE-LTL. It is symbolised by $\Diamond$ or **F**. As shown in Figure 4.2, $\Diamond$ **p** means that **p** holds sometimes in the future.

4.1.1.3 next operator

The *next* operator is a unary temporal operator in SE-LTL. It is symbolised by $\bigcirc$ or **X**. As shown in Figure 4.3, $\bigcirc$ **p** means that **p** has to hold in the next state.

[1]The OCLinEcore language provides a textual concrete syntax that makes both ECORE and OCL accessible to users., `http://wiki.eclipse.org/MDT/OCLinEcore`

[2]`http://www.acceleo.org/pages/home/en`

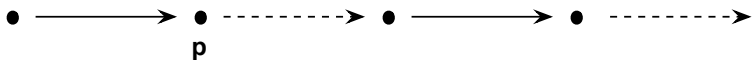

Figure 4.3 — The next temporal operator

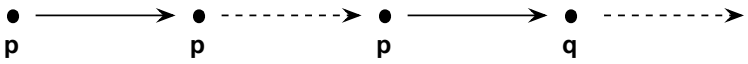

Figure 4.4 — The until temporal operator

4.1.1.4 until operator

The *until* operator is a binary temporal operator in SE-LTL. It is symbolised by **U**. As shown in Figure 4.4, **p** U **q** means that **p** holds until **q** holds.

4.1.1.5 release operator

The *release* operator is a binary temporal operator in SE-LTL. It is symbolised by **R**. As shown in Figure 4.5, **p** R **q** informally means that **q** is true until **p** becomes true, or **q** is true forever.

4.1.1.6 precedence operators

The *precedes* operator is a binary temporal introduced in [CCG+07] to ease expressing temporal properties. It is defined using the previous ones: *p precedes q = always* !(*q*) *until p*. This definition forces obtaining *p* immediately after the last *q*. We propose to define an additional operator, *before*, that has the following definition: *p before q = always* !(*q*) *until eventually p*

4.1.2 The Query Definition MetaModel (QDMM) extension

The current implementation of the *Executable DSML pattern* provides different concerns to perform model validation using graphical model animators since SDMM and EDMM introduce the required information to express the execution semantics.

In our context of model verification, three key features appear: 1) the model to be verified, 2) temporal properties to be assessed and 3) verification results. Up to now, the *Exe-*

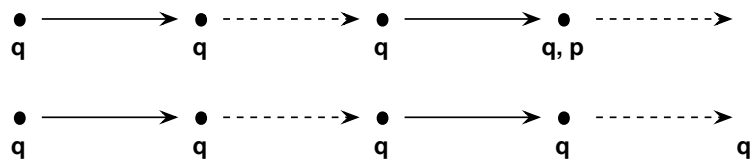

Figure 4.5 — The release temporal operator

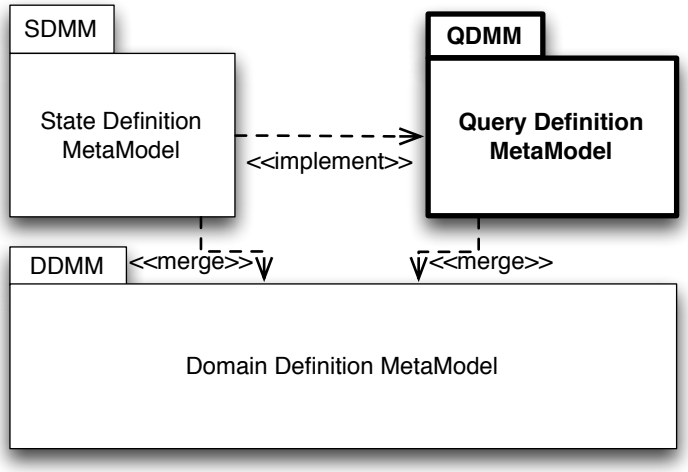

Figure 4.6 — The Query Definition MetaModel (QDMM)

cutable DSML pattern provides mandatory extensions to explicit only two features: the model and its execution. It does not offer the possibility to formalize temporal properties to be assessed.

Specifying temporal properties is typically based on the model execution which cannot be expressed using the metamodeling technologies. To ease the expression of behavioral properties, we refer to the *Property-Driven Approach* defined in [CCG+07] and experimented in [Ge14]. This approach consists in defining abstract dynamic semantics based on the properties expressed at the metamodel level.

We propose to extend the DSML metamodel to ease expressing behavioral properties. Therefore, we propose to introduce an additional extension which allows to capture different queries that can be asked on DSML conforming models. We named it *the Query Definition MetaModel* (QDMM).

Two kinds of queries are identified: primitive queries and non-primitive queries.

Primitive queries are related to the translational semantics. They are based on how the DSML designer chooses to encode DSML constructs into the formal model constructions (states, variables, etc.). They are related also to the chosen formal property language.

The non-primitive queries are related to either primitive queries or other non-primitive ones.

The QDMM is a kind of abstract view of the SDMM: it defines queries that can be asked on the DSML conforming model. SDMM may be seen as a way to implement the QDMM by choosing a set of attributes (like a Java class implements a Java interface).

Figure 4.6 shows the architecture of the QDMM extension. It is related to the DDMM by the «merge» predefined package operator and by the «implement» package operator with the SDMM.

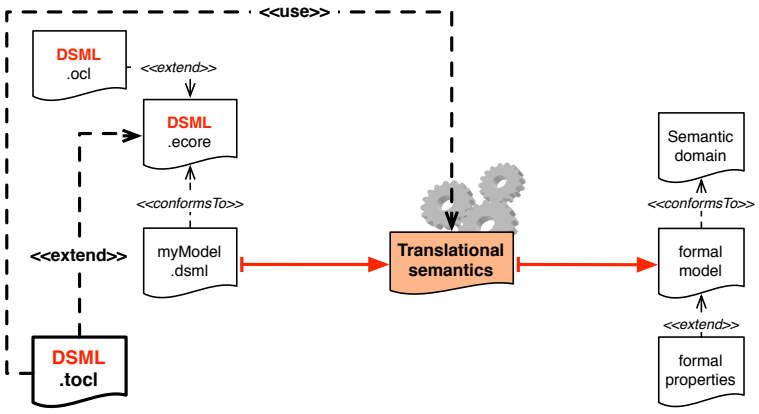

Figure 4.7 — Expressing behavioral properties on the DSML level

Once the TOCL editor is generated, it is possible to formalize DSML queries and behavioral properties. Figure 4.7 shows our approach to assist the DSML expert and the DSML designer in order to perform this task. For xSPEM, they should perform a set of steps to correctly implement SPEM queries.

1. First, the DSML expert explicits in natural language the expected behavioral properties to be assessed on SPEM models: *Does a* SPEM *model finish?* A SPEM process finishes if all its activities finish while respecting constraints imposed by dependencies and resource allocations. If a SPEM process finishes, the verification process should produce a possible terminating scenario that shows a possible execution of the SPEM process. Otherwise, it should produce a counter-example showing a deadlock that forbids the SPEM process to finish.

2. Then, based on the expected behavioral properties, he identifies informally different queries to be defined. Figure 4.8 shows the SDMM of *WorkDefinition*, obtained by applying the *Executable DSML pattern* to SPEM. It defines an attribute state that can be used to implement the queries *isStarted()* and *isFinished()* from QDMM. An additional *isFinished()* query can also be defined for *Process* meta-class.

3. Now, the DSML expert may formalize the DSML non-primitive queries and behavioral properties. Listing 4.1 shows a possible formalization of these queries using the TOCL editor. First, he imports the DSML metamodel (lines 1-2). Then, he formalizes different expected behavioral properties. Two behavioral properties are identified: The first one, *willNeverFinish*, explains that a SPEM process cannot always be finished, and the second one *willEventuallyFinish* defines that a SPEM process can eventually finish. These two behavioral properties are defined based on the *isFinished()* query on SPEM *Process* (lines 13-16). It formalizes the fact that a process is finished whether all its activities are finished. It is a non-primitive query because it relies on another SPEM query. Until now, the *isFinished()* query (lines 4-5) of *WorkDefinition* meta-class cannot be defined because it is a primitive query and it depends on the translational semantics.

```
module spem;
import "http://Spem" as SPEM

// SPEM queries
context SPEM!WorkDefinition def : isFinished(): String=
        //
        ;

context SPEM!WorkDefinition def : isStarted(): String=
        //
        ;

context SPEM!Process def: isFinished(): String =
        self.workDefinitions
            ->forAll(wd|wd.isFinished());

// SPEM behavioral properties
context SPEM!Process inv willNeverFinish:
always not self.isFinished()

context SPEM!Process inv willEventuallyFinish:
eventually self.isFinished()
```

Listing 4.1 — Formalization of SPEM queries and their related behavioral properties

4. Then, the DSML designer implements the translational semantics in order to be able to answer the queries. It maps the DSML metamodel into a semantic domain. Then, it should ease the formalization of identified primitive queries. For our running case-study explained in chapter 3, in order to verify behavioral properties on models designed on a DSML for describing processes based on SPEM, we have chosen to implement a translational semantics for SPEM into a formal semantics domain, which is time Petri nets (TPN), in order to reuse the existing model-checking tools provided by the TINA toolbox [BRV04]. Behavioral properties target the evolution of the model over time.

5. Finally, the DSML designer formalizes different primitive queries based on the implemented translational semantics. The *isFinished()* query of *WorkDefinition* meta-class returns a string which refers to the corresponding generated TPN place which characterizes the finished state for a workdefinition. A workdefinition *wd1* is finished when the corresponding place which characterizes the finished state (*wd1_finished*) obtain a token. The *isStarted()* query of *WorkDefinition* meta-class returns a string which refers to the corresponding generated TPN place which characterizes the started state for a workdefinition. A workdefinition *wd1* is started when the corresponding place which characterizes the started state (*wd1_started*) obtain a token.

   The queries on *WorkDefinition* are primitive because they are defined based on the translational semantics, whereas *isFinished()* on *Process* may be defined from the other ones. Listing 4.2 shows the implementation of these primitives queries.

```
context SPEM!WorkDefinition def : isFinished(): String=
        self.name+'_finished'
        ;

context SPEM!WorkDefinition def : isStarted(): String=
        self.name+'_started'
        ;
```

Listing 4.2 — Formalization of SPEM primtive queries

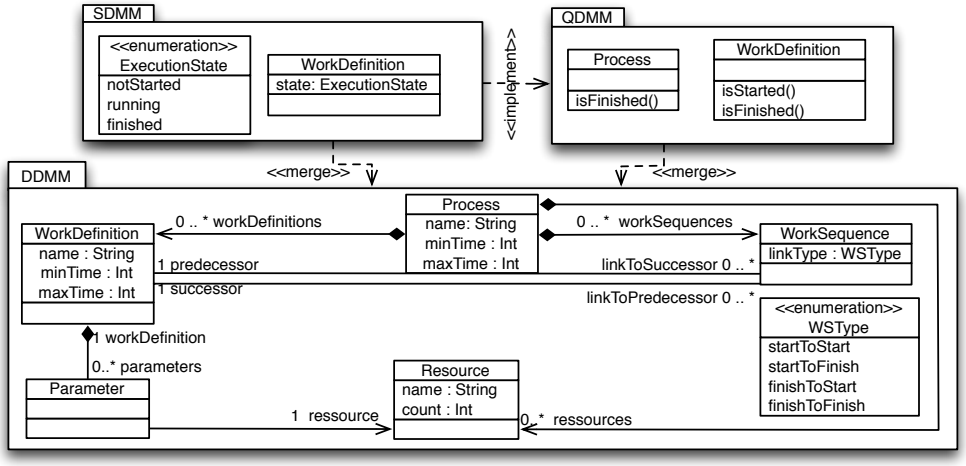

Figure 4.8 — The application of QDMM extension for XSPEM

4.1.3 Implementation

We have chosen SimpleOCL[3] as a core language for implementing our temporal extension of OCL. SimpleOCL is an embeddable OCL implementation for inclusion in transformation languages for the EMF Transformation Virtual Machine (EMFTVM).

The tool creator, Dennis Wagelaar[4], clarifies the reason for the choice of developing a new OCL tool instead of reusing one of the existing OCL tools in Eclipse[5].

He said that the different proposed implementations of OCL like MDT-OCL [Wil10] and Dresden-OCL[6] use the pivot approach which consists in defining two metamodels: the first metamodel is for parsing, the second one for representing the standard OCL metamodel. This approach makes things more complex for higher-order transformations (HOT), because it is mandatory to first transform from the concrete syntax to the pivot metamodel, do the HOT, then transform back to the concrete syntax metamodel.

This reason coincides fully with our needs. In addition, SimpleOCL meets the need to have an ATL-style implementation of an OCL editor. It is justified by the necessity to develop languages (DSLs, query languages) using higher-order transformations (HOT), the generation of a model transformation becomes easy. The similarity of the OCL part of both metamodels makes the language designer more focused on the functional part of his language. However, the OCL part will be managed with identity transformation rules.

Based on the SimpleOCL concrete syntax and metamodel, we have chosen to implement this extension using Xtext. In addition, we perform several modifications on the SimpleOCL grammar in order to support our requirements.

As SimpleOCL supports only OCL `def` definitions, we have extended it with an addi-

[3] `https://code.google.com/a/eclipselabs.org/p/simpleocl/`
[4] `http://www.micallefwagelaar.be/dennis/doku.php/start`
[5] `http://modeling-languages.com/simpleocl-tool/`
[6] `http://www.dresden-ocl.org/`

tional rule (`OclInvariant`) in order to support OCL invariants. The definition declarations allow to define queries and their related methods and the invariants allow to formalize behavioral (or temporal) properties based on the defined queries.

To introduce temporal expressions, we identify two kinds of expressions: temporal binary expressions and temporal unary expressions. The temporal binary operators, `until`, `release`, `precedes` and `before`, are defined using `BinaryTemporalOp` rule.

The second one is defined using the `UnaryOpCallExp` rule. The `UnaryTemporalOp` rule defines unary temporal operators `always`, `eventually` and `next`.

4.2 Translation of behavioral properties

Until now, we have shown what we provide for the DSML expert and designer to ease the expression of behavioral properties and their related queries in the DSML side. First, he should extend his DSML metamodel with the appropriate queries which can be assessed on DSML conforming models and then, using the developed TOCL editor, he can implement the body of these queries and their related behavioral properties. This step is mandatory but it is not sufficient. It should be completed by the necessary tooling in charge of automatically generating the corresponding formal behavioral properties.

The TOCL editor provides a high-level layer to formalize different concerns to express behavioral properties. It is based on a well-known language which is OCL. In this section, we discuss different challenges to be handled and we explain our approach to perform this translation. The formalized queries and their related behavioral properties should be translated into the formal layer. This task cannot be handled as a classical one-to-one mapping for the following reasons:

1. The gap between user and verification languages. User domain TOCL properties are expressed in order to extend the DSML abstract syntax with behavioral properties. These properties must be translated into the formal verification level (LTL for example). The gap is due to the fact that TOCL behavioral properties are expressed on the metamodeling level (in the DSML metamodel) and usually generated properties will be verified on a formal model (an instance of formal metamodel). Therefore, the TOCL behavioral properties for a DSML should be written once and verified on all DSML conforming models.

2. OCL is designed as a general-purpose language for expressing all kinds of (meta)model query and evaluating specification requirements. It allows to assess well-formedness properties. However, in our approach, OCL is extended in order to be transformed later into verification technical space. It consists of a code generation task. For example, the body of the universal quantifier `forAll` of OCL is supposed to be an OCL expression which returns a boolean value. However, in our implementation of TOCL, this kind of OCL expressions should return a string related to the chosen formal properties language. So, it is mandatory to classify what kind of iterators should be preserved and what kind of iterators should be extended.

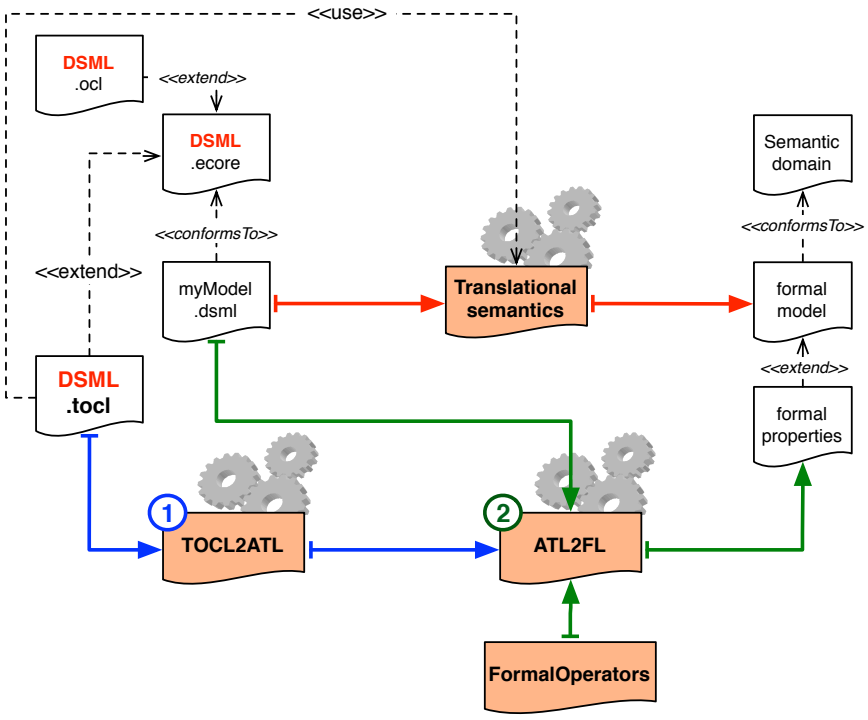

Figure 4.9 — Translating DSML behavioral properties on the formal side

3. The amount of information that must be handled is important. First, It includes the temporal aspect introduced through the extension. Second, it contains both OCL `inv` and `def` declarations. Finally, it includes primitive queries related to the translational semantics, non-primitive queries and their related behavioral properties.

4.2.1 The proposed approach to translate behavioral properties

Due to the previous reasons, the process of the generation of formal behavioral properties is more complex than a classical mapping. It relies on a higher-order transformation as illustrated on Figure 4.9 (and detailed here after).

The first transformation `TOCL2ATL` is independent of any DSML. It is a higher-order transformation that generates a model-to-text transformation, named `ATL2FL` which is specific for each DSML. `FL` suffix corresponds to the chosen formal property language. `TOCL2ATL` allows to resolve the semantic gap between both metamodeling and modeling levels. First, this transformation unpacks some OCL iterators whose body returns a boolean value in order to support defined DSML queries. Second, It converts TOCL expressions which correspond to formal formulas into OCL expressions.

Let's detail the different steps to generate the specific transformation `ATL2FL` and what is the mandatory information to finally generate formal behavioral properties. To illustrate it, we refer to our xSPEM case-study.

Dealing with OCL iterators to support QDMM Using Xtext, we have syntactically extended the OCL grammar to support temporal operators in the TOCL editor. However, it is not sufficient to generate formal behavioral properties. There are some OCL constructs which must be extended semantically. One of these ones is the iteration over collections operators that are characterized by accepting an expression as parameter and returning a boolean value.

The syntax used to call an iterative expression is the following:
`source->operation_name(iterators | body)`

- `source` corresponds to the iterated collection.
- `iterators` correspond to declared iterator variables.
- `body` corresponds to an expression applied on iterators variables .

The purpose here is not to evaluate the value of the iterating operator but to translate them to their corresponding expressions in the formal side. We are interested in iterating operators which returns a boolean value and whose `body` also returns a boolean value. Three operators are identified: `forAll`, `exists` and `one`.

The `forAll` operator is the universal quantifier of OCL. It returns a boolean value stating whether the body evaluates to true for all elements of the source collection.

The `exists` operator is the existential quantifier of OCL. It returns a boolean value stating whether the body evaluates to true for at least one element of the source collection.

The `one` operator returns a boolean value stating whether there is exactly one element of the source collection for which the body evaluates to true.

OCL provides a generic iterating operation called `iterate()`. Its syntax is shown in Listing 4.3.

```
1 source->iterate(iterator, accumulator_declaration = init_exp |
2                 body
3         )
```

Listing 4.3 — The syntax of iterate() operator in OCL

This `iterate()` operation expression has an iterator, an accumulator and a body. The accumulator corresponds to an initialized variable declaration where the resulting values are stored. The body of an `iterate()` expression is an expression that should make use of both the declared iterator and accumulator. The value returned by an `iterate()` expression corresponds to the value of the accumulator variable once the last iteration has been performed.

The `iterate()` operation is the most fundamental and complex of the loop operations. All other iterating operations can be described as a special case of `iterate()` operation [WK03]. For example, Listing 4.4 shows the implementation of the `sum()` operation which results in the sum of the elements of a set of integers using the `iterate()` operation.

```
1 Set{1,2,3}->iterate( i: Integer, sum: Integer = 0 | sum + i )
```

Listing 4.4 — The implementation of the sum() operation using the iterate() operation

When generating the higher order transformation, we rewrite the previously cited operators (`forAll`, `exists` and `one`) using the `iterate()` operation in order to be able to generate the string that corresponds to the textual syntax of this expression in the formal property language.

The universal quantifier is denoted by the logical operator symbol $\forall$. The expression: $\forall x P(x)$, denotes the universal quantification of the atomic formula $P(x)$. The expression means: "For all x, $P(x)$ holds". $\forall x$ means all the objects x in the universe. If this is followed by $P(x)$ then the meaning is that $P(x)$ is true for every object x in the universe. If the number of elements in the universe is finite then the universal quantification $\forall x P(x)$ is equivalent to the conjunction: $P(x_1) \wedge P(x_2) \wedge P(x_3) \wedge ... \wedge P(x_n)$.

Based on this notation, we decide to rewrite each `forAll` operator containing an OCL expression related to defined DSML queries into an `iterate()` expression.

Let's consider an expression using the `forAll` operator (Listing 4.5) which defines an expression related to a DSML queries.

```
1 elements->forAll(iterator | <expression-related-to-dsml-queries>)
```

Listing 4.5 — The `forAll` iterative operator syntax in OCL

This expression will be transformed into an `iterate()` expression as shown in Listing 4.6. It iterates the expression related to DSML queries on the `elements` sequence separated by the conjunction operator of the formal language chosen to map DSML abstract syntax. The first iteration consists only in printing the related expression (line 3 and then line 8) in the accumulator but the subsequent iterations consists in concatenating the accumulator with the conjunction operator and the related expression (line 5 and then line 8).

```
1 elements->iterate(iterator; accumulator : String = '' |
2    if accumulator = '' then
3         accumulator
4    else
5         accumulator + <formal-language-conjunction-operator>
6    endif
7  +
8   <expression-related-to-dsml-queries>
9 )
```

Listing 4.6 — The redefinition of `forAll` operator using an `iterate()` expression

For our xSPEM case-study, the `forAll` operator is used to express that a SPEM process is finished if and only if all its workdefinitions are finished (lines 13-16 of Listing 4.1). Listing 4.7 show a concrete redefinition of this operator with the `iterate()` operator generated from the TOCL definition of the *isFinished()* non-primitive query (lines 13-16 in Listing 4.1).

```
1 helper context SPEM!Process def: isFinished() : String =
2 self.workDefinitions
3        ->iterate(wd; wd_res : String = '' |
4                     if wd_res = '' then
5                        wd_res
6                     else
7                        wd_res + thisModule.and
8                     endif
9                     +
10                    wd.isFinished()
11 );
```

Listing 4.7 — The `iterate()` expression generated from the `forAll` operator

The second operator is the existential quantifier `exists`. It is denoted by the logical operator symbol $\exists$. The expression: $\exists x P(x)$, denotes the existential quantification of the atomic formula $P(x)$. The expression could also be understood as: "There exists an x such that $P(x)$" or "There is at least one x such that $P(x)$". $\exists x$ means at least one object x in the universe. If this is followed by $P(x)$ then the meaning is that $P(x)$ is true for at least one object x of the universe. If the number of elements in the universe is finite, then the existential quantification $\exists x P(x)$ is equivalent to the disjunction: $P(x_1) \vee P(x_2) \vee P(x_3) \vee ... \vee P(x_n)$.

Based on this notation, we decide to rewrite each `exists` operator containing an OCL expression related to defined DSML queries into an `iterate()` expression.

Let's consider an expression using the `exists` operator (Listing 4.8) which defines an expression related to a DSML queries.

```
1 elements->exists(iterator| <expression-related-to-dsml-queries>)
```

Listing 4.8 — The `exists` iterative operator syntax in OCL

This expression will be transformed into an `iterate()` expression as shown in Listing 4.9. It iterates the expression related to DSML queries on `elements` sequence separated by the disjunction operator of the formal language chosen to map DSML abstract syntax. The first iteration consists only in printing the related expression (line 3 and then line 8) in the accumulator but the subsequent iterations consists in concatenating the accumulator with the disjunction operator and the related expression (line 5 and then line 8).

```
1 elements->iterate(iterator; accumulator : String = '' |
2    if accumulator = '' then
3         accumulator
4    else
5         accumulator + <formal-language-disjunction-operator>
6    endif
7  +
8    <expression-related-to-dsml-queries>
9 )
```

Listing 4.9 — The `iterate()` expression generated from the `exists` operator

The last operator is the uniqueness quantifier. It is denoted by the logical operator symbol $\exists!$. The expression: $\exists! x P(x)$, denotes the existential quantification of the atomic formula $P(x)$. The expression could also be understood as: "There exists exactly one x such that $P(x)$". $\exists! x$ means exactly one object x in the universe. If this is followed by $P(x)$ then the meaning is that $P(x)$ is true for exactly one object x of the universe. If the number of elements in the universe is finite, then the uniqueness quantification $\exists! x P(x)$ is equivalent to:

$$P(x_1) \wedge \neg (P(x_2) \vee P(x_3) \vee P(x_4) \vee ... \vee P(x_n))$$
$$\vee P(x_2) \wedge \neg (P(x_1) \vee P(x_3) \vee P(x_4) \vee P(x_5) \vee ... \vee P(x_n))$$
$$\vee P(x_3) \wedge \neg (P(x_1) \vee P(x_2) \vee P(x_4) \vee P(x_5) \vee P(x_6) \vee ... \vee P(x_n))$$
$$\vee P(x_n) \wedge \neg (P(x_1) \vee P(x_2) \vee P(x_3) \vee ... \vee P(x_{n-1})).$$

Based on this notation, we decide to rewrite each `one` operator containing an OCL expression related to defined DSML queries into an `iterate()` expression.

Let's consider an expression using `one` operator (Listing 4.10) which defines an expression related to a DSML queries.

```
elements->one(iterator | <expression-related-to-dsml-queries>)
```

Listing 4.10 — The `one` iterative operator syntax in OCL

This expression will be transformed into an `iterate()` expression as shown in Listing 4.11. It iterates the expression related to DSML queries on `elements` sequence separated by the disjunction operator of the formal language chosen to map DSML abstract syntax.

```
elements->iterate(iterator1; accumulator1 : String = '' |
   if accumulator1 = '' then
        accumulator1
   else
        accumulator1 + <formal-language-disjunction-operator>
   endif
  + <expression-related-to-dsml-queries>
  + <formal-language-conjunction-operator>
  + <formal-language-negation-operator>
  + '('
  + elements->excluding(iterator1)
   ->iterate(iterator2; accumulator2 : String = '' |
   if accumulator2 = '' then
        accumulator2
   else
        accumulator2 + <formal-language-disjunction-operator>
   endif
        + <expression-related-to-dsml-queries>
   )
 + ')'
)
```

Listing 4.11 — The `iterate()` expression generated from the `one` operator

Handling TOCL expressions to ease the generation of formal properties In addition to the redefinition of such OCL iterators which ease the generation of behavioral properties, temporal expressions are another kind of TOCL features should also be handled in the higher-order transformation `TOCL2ATL`.

As we aim to generate formal behavioral properties, it is mandatory to handle expressions which combine DSML queries and temporal expressions (binary and unary expressions). These expressions should be translated into an OCL conjunction expression where the operator is translated into the use of the corresponding formal operator in the `FormalOperators` library.

Listing 4.12 shows an abstract view of a temporal unary expression formalized using our TOCL editor and Listing 4.13 shows the generated expression.

```
<temporal-operator> <unary-expression>
```

Listing 4.12 — A unary temporal expression

```
<formal-language-temporal-operator> + <redefined-unary-expression>
```

Listing 4.13 — A translation of a TOCL unary temporal expression into a string concatenation in OCL

For our XSPEM case-study, the temporal expressions are used to define temporal invariants. Line 19 of Listing 4.1 shows a temporal expression that specifies that a SPEM model will never finish. Listing 4.14 shows the generated concatenation expression. `always`, `not`, `LeftBrace` and `RightBrace` refer to different attributes defined in the formal operators library.

```
thisModule.always
+ thisModule.LeftBrace
+ thisModule.not + thisModule.LeftBrace + self.isFinished() + thisModule.RightBrace
+ thisModule.RightBrace
```

Listing 4.14 — The *willNeverFinish()* invariant generated in the ATL level

In addition, we handle boolean expressions which should be transformed into string concatenation expression. Based on the returned type of an operand, we decided to rewrite the corresponding expression in the generated `ATL2FL` transformation. This targets only boolean operators: the negation operator `not`, the conjunction operator `and`, the disjunction operator `or` and the implication operator `implies`.

For our SPEM case study, once the DSML designer formalizes his DSML queries and their related behavioral properties based on the defined translational semantics which targets the TPN technical space and the TINA toolbox, we perform the higher-order transformation `TOCL2ATL`. It generates a model-to-text transformation `ATL2LTL` (`LTL` is the chosen formal property language) shown in Listing 4.15.

```
query SPEM_Properties = thisModule.generateLTL().writeTo('SPEM/BehavioralProperties.ltl');

uses LTLOperators;

helper context SPEM!WorkDefinition def: isFinished() : String =
 self.name + '_finished';

helper context SPEM!WorkDefinition def: isStarted() : String =
 self.name + '_started';

helper context SPEM!Process def: isFinished() : String =
 self.workDefinitions
)->iterate(wd; wd_res : String = '' |
  if wd_res = '' then
   wd_res
  else
   wd_res + thisModule.and
  endif + wd.isFinished()
 );

helper context SPEM!Process def: willNeverFinish() : String =
'op_willNeverFinish_' + SPEM!Process.allInstances()->indexOf(self).toString() + '_=_' +
thisModule.always + thisModule.LeftBrace + thisModule.not + thisModule.LeftBrace + self.isFinished()
+ thisModule.RightBrace + thisModule.RightBrace + ';_\n'
+
'willNeverFinish_' + SPEM!Process.allInstances()->indexOf(self).toString() + ';';

helper context SPEM!Process def: willEventuallyFinish() : String =
'op_willEventuallyFinish_' + SPEM!Process.allInstances()->indexOf(self).toString() + '_=_'
+ thisModule.eventually + thisModule.LeftBrace + self.isFinished() + thisModule.RightBrace + ';_\n'
+
'willEventuallyFinish_' + SPEM!Process.allInstances()->indexOf(self).toString() + ';';

helper def: generateLTL() : String =
SPEM!Process.allInstances()->collect(instance_Process |instance_Process.willNeverFinish())->flatten()
        ->iterate(input_Process; res_Process : String = '' |res_Process + input_Process+'\n')
+
SPEM!Process.allInstances()->collect(instance_Process |instance_Process.willEventuallyFinish())->flatten()
        ->iterate(input_Process; res_Process : String = '' |res_Process + input_Process +'\n');
```

Listing 4.15 — The generated `ATL2LTL` transformation

Let's detail different parts of this transformation. First, it uses a predefined `LTLOperators` library which contains LTL operators as ATL attributes (`FormalOperators` in Figure 4.9). Next, it contains a set of helpers which correspond to DSML queries: *isFinished()* (lines 4-5) and *isStarted()* (lines 6-7) for *WorkDefinition*, and *isFinished()* for *Process*. Then, there are two helpers which correspond to TOCL invariants: *willNeverFinish()* (line 19) and *willEventuallyFinish()* (line 25). Finally, a global helper named *generateLTL()* (line 31) concatenates different helpers which correspond to TOCL invariants. It is the entry point helper of this query (line 1). This one takes a DSML conforming model and generates its corresponding formal properties.

4.2.2 The generation of formal properties

Running the higher-order transformation `TOCL2ATL` has allowed to generate the second transformation `ATL2FL`. This last model-to-text transformation is an ATL query. It imports the `FormalOperators` library defined previously.

The formal operators library The last two subsections have shown the core of the higher-order transformation `TOCL2ATL` that generates another transformation `ATL2FL`. As we aim giving the DSML designer a higher-order tools to implement verification tasks for a DSML, it is mandatory to define a library which contains formal operators as ATL attributes. An ATL attribute can be viewed as a constant. It defines the concrete syntax of formal operators given by the chosen verification toolkit.

For example, listing 4.16 shows the `always` attribute defined for the TINA toolbox and its SELT model-checker.

```
1 helper def: always : String= '␣[]␣';
```

Listing 4.16 — An `always` attribute

The generated transformation `ATL2FL` accepts a DSML conforming model and generates behavioral properties in the formal side. Taking the SPEM model defined in the Figure 2.2, the corresponding generated LTL properties are shown in Listing 4.17.

```
1 op willNeverFinish =
2 [](-(Designing_finished /\ Documenting_finished /\ Programming_finished /\ TestCaseWriting_finished));
3 willNeverFinish;
4
5 op willEventuallyFinish =
6 <> (Designing_finished /\ Documenting_finished /\ Programming_finished /\ TestCaseWriting_finished);
7 willEventuallyFinish;
```

Listing 4.17 — The generated LTL properties

4.3 Related works

The problem of the specification and verification of behavioral properties for DSMLs has been widely addressed by the software engineering community. Different approaches are proposed to ease the expression of behavioral properties.

To verify BPEL service composition schemas, [YMH+06] proposes a property specification language based on ontologies and named PROPOLS and an associated approach to the verification of BPEL schemas. This approach allows composition of the patterns defined by Matthew Dwyer in [DAC98]. These patterns are close to TOCL temporal operators and composition corresponds to OCL operators. To guide the semantic mapping of PROPOLS properties, the authors choose to append directly semantic annotations to the WSDL file. For the verification process, a semantic equivalent Total and Deterministic Finite State Automata (TDFA) is built for every pattern property and for the BPEL schema, a finite and deterministic Labeled Transition Systems model is generated. Finally, the compliance of the BPEL schema to the PROPOLS properties is then checked.

The proposed approach allows to specify high-level properties conforming to Dwyer patterns. Compared to our approach, it can be classified as a specific approach focusing on BPEL schemas. To adapt it to another modeling language, it is mandatory to express requirements on the semantic mapping. In our approach, the QDMM provides a generic metamodeling extension which allows to capture different possible queries asked on a DSML conforming model.

In [MWVD13], an approach named *ProMoBox* is proposed. It assists the DSML engineer in the specification and verification properties at the DSML level. The *ProMoBox* contains three metamodels: the first one allows to specify the quantification which can be the universal one or the existential one. The second metamodel is the temporal pattern which is defined based on the Dwyer's specification patterns. The third one is the structural pattern which allows to query a model. The approach consists in matching the DSML classical metamodel with the three languages of *ProMoBox*. Therefore, a specific property language is generated for each DSML. Then, the *ProMoBox* model is transformed into LTL formulas using model-to-text transformation. They don't refer to the translation of the design model into Promela. The authors state that the current implementation of their approach can be shown as very near to the formal side. They aim to go one step higher in the abstraction level by providing a DSML for their use-case. In addition, it is necessary to connect each DSML to the *ProMoBox*. For us, we separate the design level and the implementation one. Our contribution, the QDMM extension, replies to this need by defining an additional information can be asked on a design model and the TOCL editor allows to formalize behavioral properties. In [MDL+14], the *ProMoBox* approach was extended with additional sub-languages annotated with the runtime information that eases the generation of LTL properties from a *ProMoBox* specification.

The cited approaches aim to express high-level behavioral properties. However, the proposed technologies requires additional knowledge for a designer who is already familiarized with OCL technologies. We think that the TOCL implementation is easier for the DSML designer introducing different required elements to express behavioral properties in the metamodel level.

5 Feedback verification results

Résumé

Nous avons proposé dans le chapitre précédent un langage pour exprimer des propriétés sur les DSMLs qui sont automatiquement traduites en propriétés formelles pour être vérifiées par les outils de model-checking. Dans ce chapitre, nous abordons la remontée des résultats de la vérification.

En effet, notre objectif est que l'utilisateur final du DSML qui n'est pas à avoir de connaissance particulière des langages et outils de vérification s'attend à une approche transparente qui cache et intègre de manière transparente les outils associés dans des chaînes d'outils de vérification tout en profitant des avantages de ces méthodes puissantes. Aussi, il est nécessaire de traduire les résultats obtenus au niveau formel en résultat au niveau du domaine de l'utilisateur (le DSML). Le défi consiste donc à tirer le meilleur des outils formels de sorte que l'utilisation ne soit pas affaiblie par les aspects formels non cachés.

Par conséquent, une tâche essentielle pour les concepteurs d'un DSML est de remonter les résultats de la vérification générés par les outils de model-checking. La remontée des résultats de vérification rend l'utilisation des méthodes formelles plus prometteuse pour les utilisateurs finaux.

Dans la littérature, ce problème est soit non traité, soit il est résolu par des solutions codées en dur ou ad-hoc. Par conséquent, dans cette partie, nous proposons une approche pour résoudre ce problème en fournissant, à la fin, une solution générale d'un outil générique.

Tout d'abord, nous expérimentons la définition d'une transformation dans le sens inverse (du formel vers le DSML) selon les extensions du patron de métamodélisation appliquées sur les deux niveaux. Ensuite, nous avons expérimenté les transformations de modèle bidirectionnelles. Ce type de transformation permet de combiner à la fois la sémantique translationnelle avec la gestion de retour des résultats de vérification.

Nous appliquons ces deux approches dans notre étude de cas de la vérification des modèles SPEM en utilisant les TPN. Nous discutons aussi de la généralité de ces approches et le genre de solution qui devrait être proposée pour obtenir une solution générique. Enfin, nous présentons notre langage dédié proposé qui permet de définir séparément les correspondances entre les événements du niveau DSML et leurs correspondants au niveau formel.

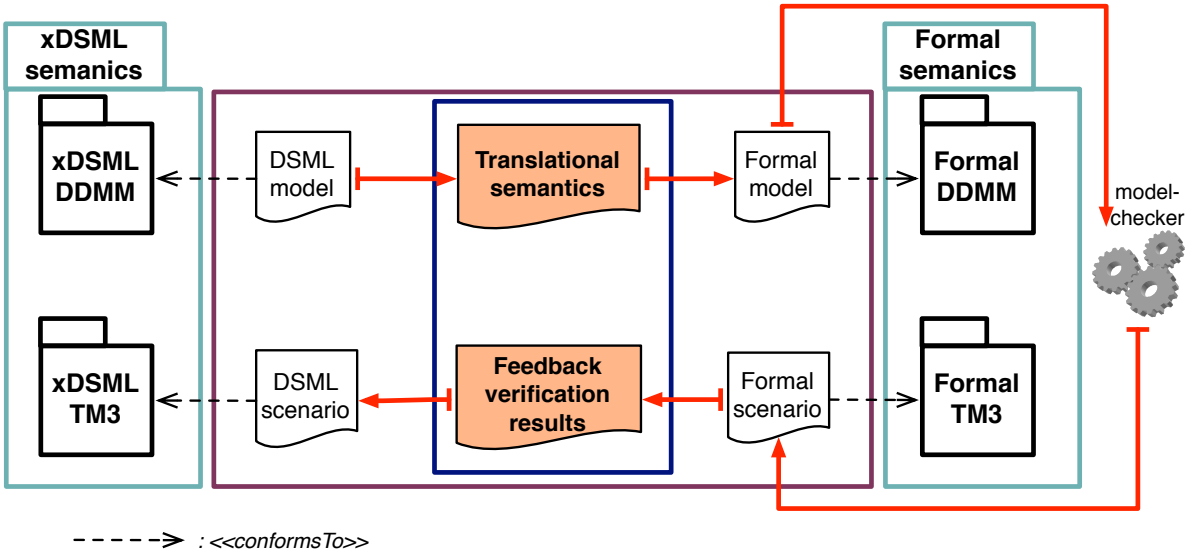

Figure 5.1 — Overview of the feedback of verification results in the DSML V&V context

WE have proposed in the previous chapter a language to express properties on a domain specific model which are automatically translated into formal ones and checked by the tools. In this chapter, we address the feedback of verification results.

Indeed, DSML end-user who is familiar with DSML concepts should not be required to have a solid knowledge on formal languages and associated tools. The challenge is thus to leverage formal tools so that the use is not burdened with formal aspects.

This one expects a seamless approach which hides and embeds transparently the associated tools in automated verification toolchains while enjoying the benefits of these powerful methods. Therefore, one critical task for the DSML designers is getting back verification results generated by the model-checking tools. The feedback of verification results make the use of formal methods more hopeful for end-users.

Figure 5.1 shows the expected framework to integrate a hidden formal verification approach for a new DSML. It extends the translational semantics which maps a DSML conforming model into a formal model with the feedback of verification results from the formal side into the DSML one.

It is mandatory to ease for the DSML designer the feedback of verification results and assist him to generate a DSML verification framework while reducing development costs. However, in the literature, this problem is either not handled or it is resolved by hard-coded or ad-hoc solutions. Therefore, in this part, we propose an approach to overcome this problem by providing in the end a general tool-independent solution.

First, we experiment the definition of a backward transformation based on the executable extension from the *Executable DSML pattern* introduced both on the DSML and formal sides.

Second, we have experimented bidirectional model transformations. This kind of transformation allows to combine the translational semantics with the management of the feed-

back of verification results.

We apply both approaches within our case-study of the verification of the SPEM conforming models using TPN. We discuss also about the generality of these approaches and the kind of solution that should be proposed to obtain a generic solution. Finally, we show our proposed domain-specific programming language (DSPL) which allows to define separately the mapping between DSML events and their corresponding ones at the formal level.

5.1 Defining a backward transformation

To feedback the verification results to the DSML level, we have initially chosen to define a model-based toolchain to manage verification results generated by the TINA toolbox. Figure 5.2 illustrates this approach: the top of the figure recall the translational semantics and the bottom explains how verification results are analysed to feedback results at the DSML level.

Using Xtext, we analyze the output of the SELT model-checker and produce a TPN scenario (*results.tpnscn* in Figure 5.2) which conforms to the TM3 extension of the TPN metamodel.

Listing 5.1 gives the TPN scenario corresponding to the verification output generated by the SELT model-checker shown in Listing 3.4. This scenario is an instance of the TM3 and EDMM metamodeling extensions defined at the TPN level. It shows a succession of TPN events which consist in firing TPN transitions (instances of the *FireTransitionEvent* meta-class in the EDMM of TPN).

```
1 FireTransitionEvent Designing_start
2 FireTransitionEvent Designing_finish
3 FireTransitionEvent Documenting_start
4 FireTransitionEvent Documenting_finish
5 FireTransitionEvent TestCaseWriting_start
6 FireTransitionEvent Programming_start
7 FireTransitionEvent Programming_finish
8 FireTransitionEvent TestCaseWriting_finish
```

Listing 5.1 — A TPN scenario corresponding to the verification results generated by the SELT model-checker shown in Listing 3.4

The TPN scenario is then transformed to an XSPEM scenario. The transformation (*TPNSCN2SPEMSCN.atl* in Figure 5.2) converts transition firing events *FireTransitionEvent* to XSPEM events, either start (*StartWD*) or finish (*FinishWD*) a *WorkDefinition*. The naming conventions are used to decode the fired transition names and produce the corresponding XSPEM events and their target workdefinitions.

Listing 5.2 shows an ATL rule (PNEventToStartWD) which allows to produce a XSPEM *StartWD* event from a TPN *FireTransitionEvent* one. It consists in selecting a TPN *FireTransitionEvent* instance whose name ends with `'_start'`, to compute a substring prefix of its name (line 11), and, finally to select the workdefinition whose name corresponds to the substring (line 12).

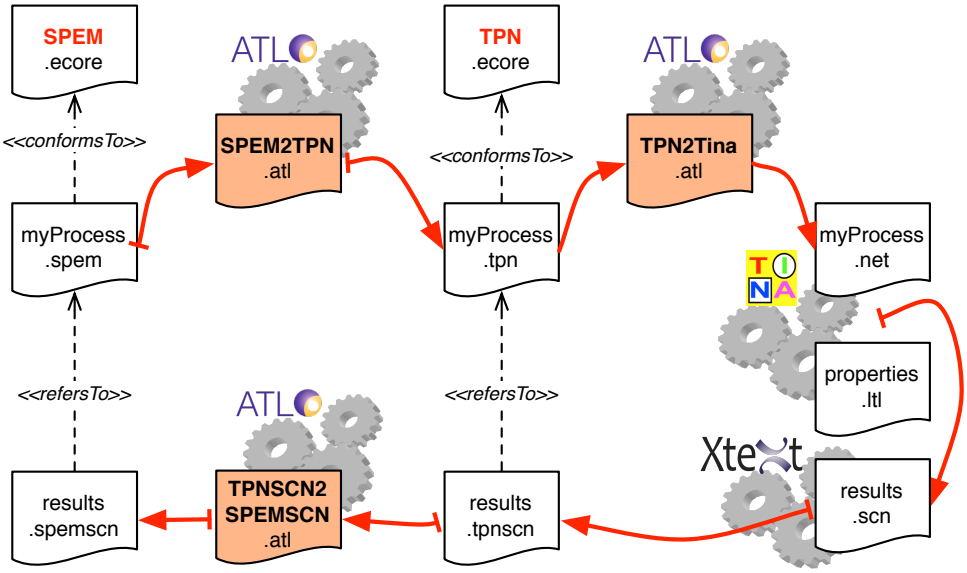

Figure 5.2 — Feedback verification results generated by the TINA toolbox into the SPEM level

```
1  rule PNEventToStartWD{
2          from
3                  tpn_ev : TPNScenario!FireTransitionEvent(
4                          tpn_ev.name.endsWith('_start')
5                  )
6          using{
7                  startIndex : Integer = tpn_ev.name.indexOf('_start');
8          }
9          to
10                 startWD : SPEMScenario!StartWD(
11                         name <- tpn_ev.name.substring(1,startIndex),
12                         workdefinition <- thisModule.getWorkDefinition(startWD.name)
13                         )
14 }
```

Listing 5.2 — An ATL rule to generate a *StartWD* event from a *FireTransitionEvent* one

The XSPEM scenario corresponding to the previous TPN one is shown on Listing 5.3.

The main goal of our work consists in facilitating the feedback of verification results generated from the model-checker. Therefore, we refer to the *Executable DSML pattern* which augments the DSML abstract syntax with behavioral aspects in order to capture runtime information. In this work, we decide to encode manually the feedback transformation. This backward transformation can be not obvious and thus include errors. Therefore, we propose, in the next section, to unify the two transformations to achieve feedback by backward evaluation of the unified bidirectional model transformation.

```
1 StartWD Designing
2 FinishWD Designing
3 StartWD Documenting
4 FinishWD Documenting
5 StartWD TestCaseWriting
6 StartWD Programming
7 FinishWD Programming
8 FinishWD TestCaseWriting
```

Listing 5.3 — A SPEM scenario corresponding to the TPN scenario shown in Listing 5.1

5.2 The use of bidirectional transformation

Bidirectional model transformations are an appropriate candidate [Ste08] to feedback automatically verification results from the formal side to the DSML. It is a mechanism that maintains the consistency between two (or more) artifacts [CFH+09, HSST11]: when one evolves, the other is updated to reflect the changes.

Bidirectional model transformations aim to unify the two unidirectional transformations into one bidirectional definition. In fact, the translational semantics can be performed due to the forward evaluation and the feedback of the verification results is achieved by backward evaluation of the unified transformation.

We reuse the taxonomy proposed in [FGM+07] which consists of the forward transformation *get* to obtain target artifact t from source artifact s, and backward transformation *put* from the pair of updated target t' and original source s, to obtain updated source s'.

According to Figure 5.1, the source model in this framework is the model conforming to a DSML DDMM. We identify it as a DSML *Model*. The translational semantics defined in the DSML verification context is shown as the *Forward transformation* in the bidirectional model transformation framework. This transformation allows to map the DSML *Model* into a formal one in order to generate a formal model understood by model-checkers. It corresponds to the target model in the bidirectional model transformation framework. It is shown as a *Formal Model*. Verification of models using model checking provides execution results only in the target technical space when the verification fails.

Updating the target model in the bidirectional model transformation framework, corresponds, in the DSML model verification context, to produce the verification information in the formal target model (*Formal scenario*). The *Backward transformation* allows to feedback runtime information into the DSML level. A DSML *scenario* is generated.

The overviews of both approaches are broadly similar in terms of model and manipulated information. This eases the use of bidirectional model transformation in the DSM verification context.

To adopt this solution, we introduce a collaboration framework between the GROUNDTRAM bidirectional model transformation system [HHI+11, HHI+13] and the *Executable DSML pattern*.

5.2.1 Bidirectional Model Transformation with GROUNDTRAM

GROUNDTRAM is a well-behaved, language-based (functional), compositional (allows intermediate models), text based bidirectional modeling framework based on bidirectionalized graph transformations [HHI+10]. It has been actively used in several fields, including recent attempts in co-evolution of models and code in software engineering [YLH+12], as well as in a collaboration framework for model development in synthetic biology [WKH13].

A notable alternative graph transformation based framework would be those based on Triple Graph Grammars [SK08]. We have chosen GROUNDTRAM for the ease of compositions in transformations.

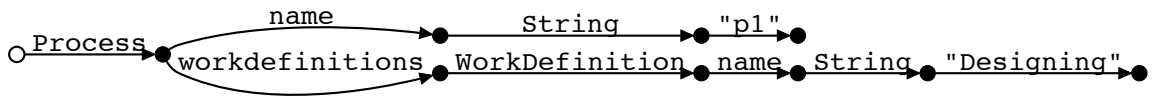

Figure 5.3 — An edge-labelled graph for a SPEM model

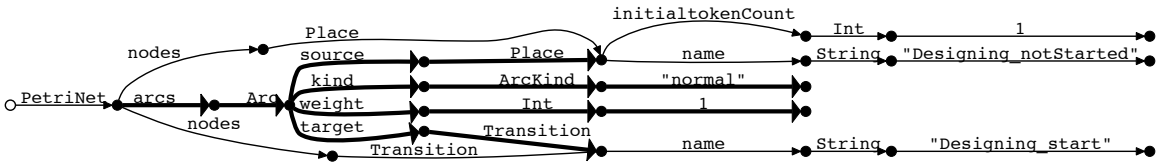

Figure 5.4 — An edge-labelled graph for a TPN model

We introduce in the rest of this section the features of GROUNDTRAM that are relevant to our approach.

5.2.1.1 Data Model

Graphs that represent models in GROUNDTRAM are rooted and edge-labelled. All the information is stored in edge labels and node identifiers have no particular meanings. They are just unique identifiers. Outgoing edges of a node are not ordered.

The graph in Figure 5.3 represents a subset of the SPEM model in Figure 2.2 with only one workdefinition "`Designing`" and the graph in Figure 5.4 represents a corresponding TPN model with just one place and one transition and an arc between them. Models are represented in a modular way. For example, nodes and arcs of the TPN model are represented by subgraphs under edges labelled `nodes` and `arcs`, respectively.

5.2.1.2 Bidirectional Transformations

Transformations in GROUNDTRAM are written in UnQL+, an extension of select-where style graph query language UnQL[BFS00] with constructs for replacement, extension and deletion of subgraphs specified by regular expressions over paths from the root.

The forward interpretation (*get*), if given the graph of SPEM in Figure 5.3 as its input (bound to global variable $*db*), outputs the TPN in Figure 5.4 (*without* the bold part), where **where** part (selectively) extracts labels or subgraphs using regular path patterns and binds them to variables (the name of the workdefinition "`Designing`" under the path `Process.workdefinition.....String` from the root is bound to $*name*), and **select** part constructs the results using these bindings (subgraphs encoding places and transitions are constructed using the names derived from the workdefinition name).

After the forward transformation, user can modify the resulting target. The backward transformation takes this updated target as well as the original source to propagate these changes back to the source.

The target updates are propagated via variable bindings. The source is given as a binding

```
select {PetriNet:{nodes:{Place: {name:{String:{$name^"_notStarted":{}}},
                                 initialtokenCount:{Int:{1:{}}}},
                        nodes:{Transition:{name:{String:{$name^"_start":{}}}}}}}
where {Process.workdefinitions.WorkDefinition.name.String:{$name:{}}} in $db
```

Listing 5.4 — UnQL+ transformation from SPEM to TPN

of variable $*db*. The forward transformation interprets the UnQL+ program under this binding, which is extended by the bindings introduced by **where** and related clauses. Backward transformation updates these bindings according to the updates of the target.

GROUNDTRAM supports back propagations of edge-renaming, edge deletions and subgraph insertions by separate algorithms (commands).

Note that several edge renamings may have to be propagated by separate backward transformations. For example, suppose we simultaneously edit two subgraphs that are produced via different subexpressions in the transformation that share a variable. Then during the backward transformation, two instances of updated bindings of the variable are produced and merged [HHI+10]. If both of the bindings are different, an error reporting inconsistent updates is signaled.

Bidirectional transformation usually propagates updates uniquely, meaning that user does not have much control on the propagation in case of multiple possibilities. However, users can design transformations that ease the intended propagation. In our work, propagation of verification results is achieved by extending the original transformation to embed subgraphs that encode event sequence information as shown in the next subsection.

5.2.2 Combining the *Executable DSML pattern* with the GROUNDTRAM framework

Combining these two different approaches may encounter some obstacles. This is due to the difference in the "nature" of the exchanged information. In the *Executable DSML pattern*, the information is modeled using Eclipse Modeling Framework (EMF) [SBPM08] with XMI Schema, but the GROUNDTRAM platform is used to do bidirectional transformation for graphs.

This gap between these two different levels can affect the expressiveness of the exchanged information. Taking an example in Figure 5.5, we show an edge-labelled graph that defines an XSPEM process graph conforming to a subset of the SPEM model shown in Figure 2.2. It only contains two workdefinitions *Programming* and *TestCaseWriting* and two worksequences between them. This SPEM process is extended with a scenario which is composed of a set of events (in the dashed rectangles). It represents the scenario shown in the end of Listing 5.3 (lines 5-8). In our approach, this graph corresponds to the *Updated* DSML *Source Model*. The *Process* sub-graph is the source model and the *Scenario* sub-graph is the information propagated from the modified formal target model.

Right now, we can say that the integration of *Executable DSML pattern* in the GROUNDTRAM framework is advantageous and suitable. However, an important information is miss-

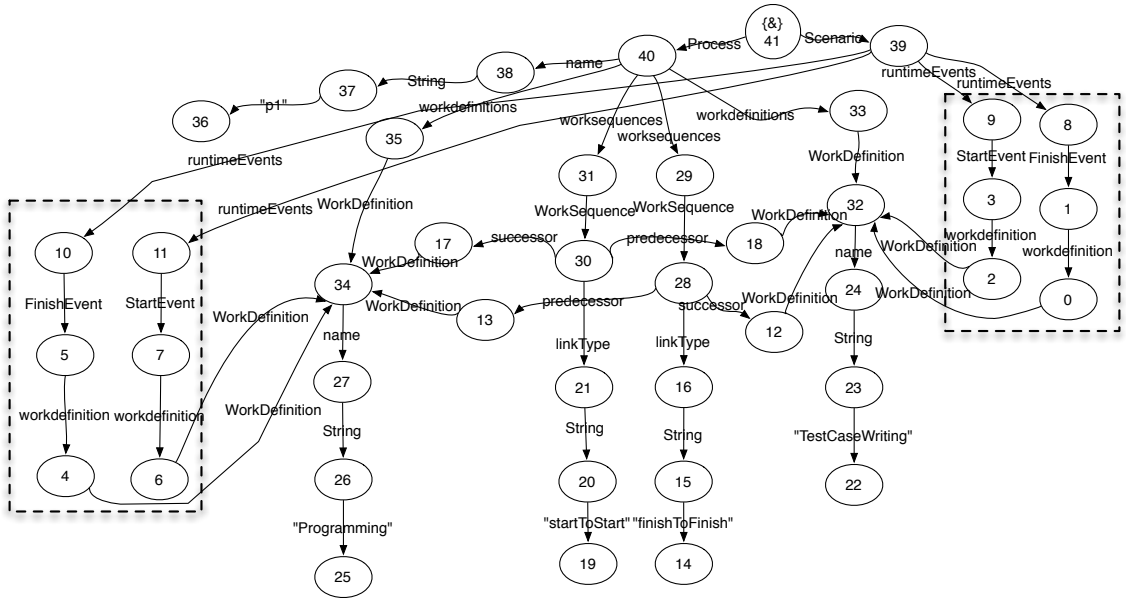

Figure 5.5 — An edge-labelled graph for an XSPEM process enriched with verification results

ing which is the order of events. Using this representation, we can show events which compose the scenario but we cannot identify which is the first event, the second event, etc.

To resolve this major problem, the application of *Executable DSML pattern* should be extended to capture the order. Two kinds of solution are proposed: the first one consists in adding a *next* reference in the `RuntimeEvent` meta-class of the TM3 to itself and the second solution consists in adding an attribute in the same meta-class. We choose to add an integer attribute in the DSML `RuntimeEvent` meta-classes, named "rank", to capture events' order in the DSML scenario because it is more easy in the backward transformation to rename an edge than to add another one. Consequently, this extension in the DSML level, should also be defined in the formal level in order to capture the rank information in the formal level and after feedback of this information into the DSML level.

These extensions are done in both the DSML (XSPEM EDMM) and the formal levels (TPN EDMM) to capture the order of events. Now, we can manipulate graphs which contain the complete information. This allows to use the *Executable DSML pattern* in the GROUNDTRAM framework consistently.

So, according to the *Executable DSML pattern* applied into the SPEM standard, the idea consists of creating, for each workdefinition, two events ("StartWD" and "FinishWD") that are initialized with rank =0. The update of this information corresponds to modifying the rank value of an XSPEM event with the rank value of the corresponding TPN event. This operation entails renaming edges that encode rank attributes of several events in the generated scenario simultaneously. However, as we said in the subsection 5.2.1, simultaneous edge-renamings may signal an error reporting inconsistent updates during the backward transformation.

Therefore, we propose to apply and propagate the modification of these different rank

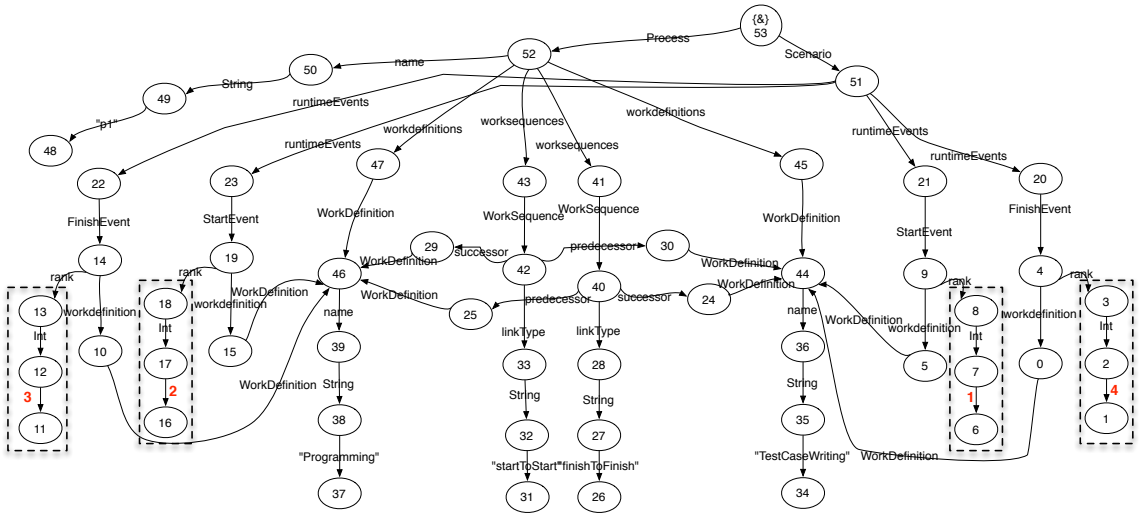

Figure 5.6 — An edge-labelled graph for an XSPEM process enriched with verification results and adapted into the *Executable DSML pattern*

attribute values step by step. Thus, after generating the formal scenario with the model checker, we 1) modify the earliest event in the formal model, 2) propagate this modification into the DSML model using the backward transformation, 3) take the updated DSML source model as a new DSML source model, 4) re-transform it into a formal target model and return to step 1) until the whole scenario has been propagated. This method allows not only to feedback the complete scenario, but also to simulate the execution of the model.

Figure 5.6 shows an edge-labelled graph that defines an XSPEM process that represents the XSPEM model extended with a complete scenario which is composed of a set of events whose rank attribute values are updated with the indexes of events in the scenario (in the dashed rectangles). They represent the scenario shown in Listing 5.3.

5.2.3 Implementation

In this subsection, we introduce the implementation of our approach using the GROUNDTRAM framework.

Once the bidirectional transformation XSPEM2TPN.unql+ is defined, it should be completed with the mandatory tooling to integrate DOT models, the native representation format in the GROUNDTRAM toolset, with model verification tooling and especially TINA toolbox. Figure 5.7 shows the complete overview of our approach. The generated DOT formal target model is parsed using an existing DOT Xtext grammar[1]. An XMI formal target model is generated. Using a model-to-text ATL transformation, a textual TPN is built to be used by the SELT model checker of the TINA toolbox in order to verify whether the temporal property is verified. If not, a counter example is generated. Using a model-to-model ATL transformation, we update the XMI formal target model and it generates the XMI formal modified target model. Next, we define a model-to-text ATL transformation to produce the DOT formal

[1] `http://code.google.com/p/emfmodelvisualizer/source/browse/galileo/trunk/org.openarchitectureware.vis.graphviz/src/org/openarchitectureware/vis/graphviz/Dot.xtext`

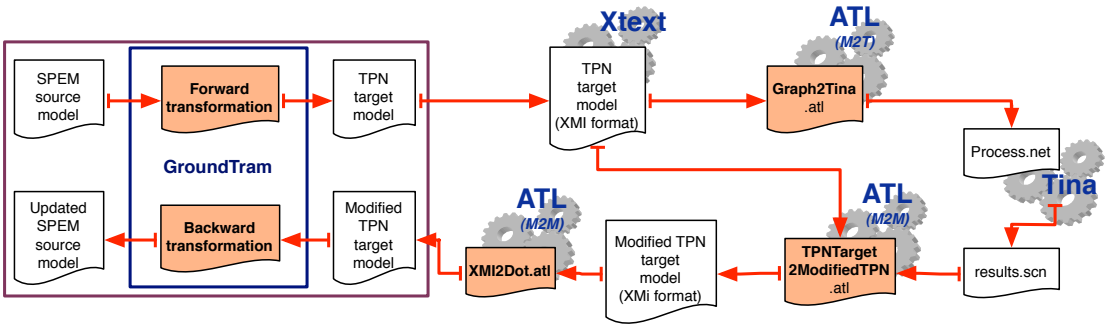

Figure 5.7 — A complete overview of the integration of the *Executable DSML pattern* with GROUNDTRAM framework in the context of DSML V&V

modified target model. Finally, using the backward evaluation of our XSPEM2TPN.unql+ transformation, the DOT DSML updated source model is generated.

5.2.4 Synthesis and discussion

The adopted approaches to ease the feedback of verification results from the formal level after performing model-checking to the DSML level consist in either defining a backward transformation or unifying both transformation into one bidirectional transformation.

Based on both EDMM extensions, these approaches allow to define a kind of mapping between both levels using model transformations techniques. First, The backward transformation takes a formal scenario as a succession of formal events and using naming conventions, a corresponding scenario in the DSML level is generated. Second, we proceed to bidirectional transformations which allow to define a well-behaved bidirectional transformation combining the translational semantics and the manage of verification results. These proposed solutions are still ad-hoc and do not resolve the main problem for the DSML designer which is obtaining a generic suitable tool to proceed the feedback of verification results. In the next section, we introduce a language to express additional information that ease the feedback of verification results.

5.3 FEVEREL: Feedback Verification Results Language

We have shown in the two previous sections, possible solutions to deal with the feedback of verification results into the DSML level. According to the Figure 5.1, the first approach encodes manually the backward transformation and the second one shows that we can combine both the translational semantics with the feedback of the verification results in a bidirectional transformation.

We have experimented the generation of the backward transformation from the translational semantics. This solution consists in analysing the ATL implementation of the translational semantics [ZCP13b]. The drawback is that it is strongly coupled to ATL and forbid to use another transformation language.

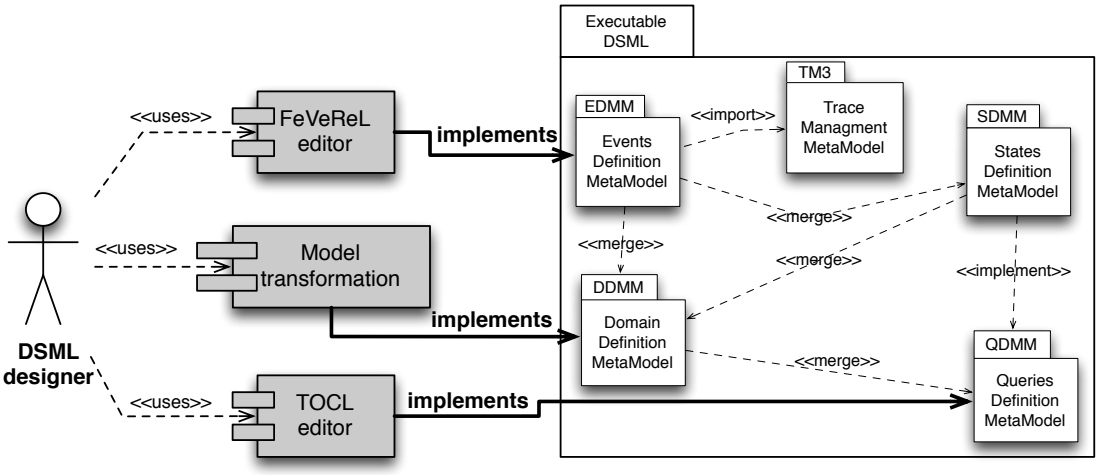

Figure 5.8 — Towards the generation of a DSML verification framework

The previous approaches do not answer to the DSML designer needs. They are ad-hoc approaches and depend especially on the used technology to implement the translational semantics. Therefore, it is mandatory to offer the DSML designer a high-level abstraction with the appropriate tooling which allows to manage the feedback regardless of how the translational semantics is implemented and, then, generate the model transformation which feedbacks the verification results.

In this section, we will introduce our proposal to ease, for the DSML designer, the feedback of verification results from the formal to the DSML levels. It is the Feedback Verification Results Language (FeVeReL). We start by showing the motivation to develop it and then its architecture. Next, we show adopted patterns to implement our DSPL and how they can ease its support. Next, we detail different DSPL elements: its abstract syntax, its textual concrete syntax and its semantics. Finally, we conclude this section by an application of our DSPL on the case-study of the verification of SPEM conforming models using TPN.

5.3.1 Motivations

One of key tasks for the definition of useful V&V activities for new DSMLs is feeding back to the DSML end-user verification results generated in the formal side by model checking tools. The DSML designer should guarantee this function. It consists in generating a scenario showing the execution of the model at the DSML level.

Actually, the DSML designer can define a model-to-model transformation which injects a formal scenario and generates a DSML scenario. However, this transformation can be complex and the DSML designer may fail to develop it because the mapping between events of both sides can be sophisticated. Also, as we aim to use the *Executable DSML pattern* to explicit different DSML concerns and the TOCL editor to define queries for a DSML based on the defined translational semantics, it is more suitable to go on in the same way by providing a DSPL based on OCL to explain how DSML events can be observed in the formal

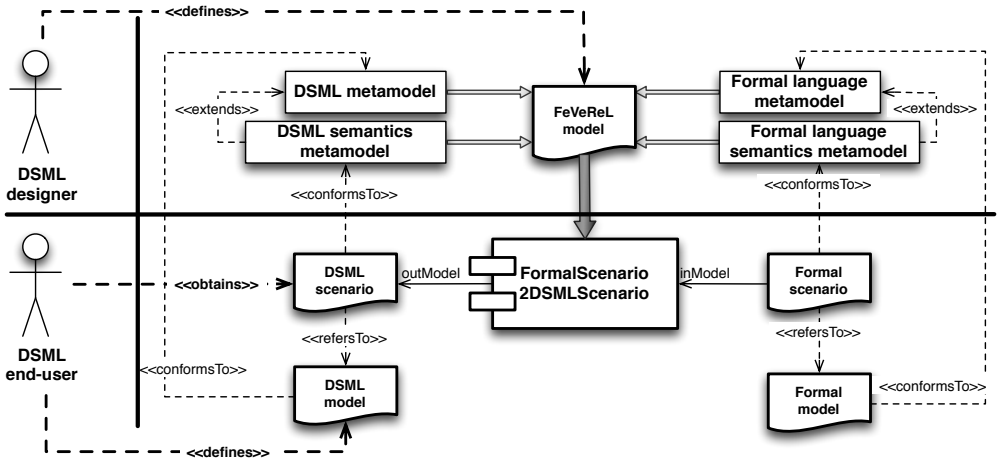

Figure 5.9 — Architecture of FEVEREL

side.

Figure 5.8 shows our approach to ease the DSML designer tasks to generate a complete DSML verification framework for the DSML end-user. First, using model transformation techniques, The DSML designer implements the translational semantics which allows to map DSML abstract syntax (DSML DDMM) into a semantic domain. Next, based on the defined translational semantics, he implements the DSML queries (DSML QDMM) using the TOCL editor in order to automatically generate formal ones. Finally, to get back verification results generated in the formal side, the DSML designer has to define mappings between DSML events modeled in the DSML EDMM and their corresponding ones in the formal side.

The adopted approach aims to separate the implementation of different concerns for the DSML designer. Each element of the DSML metamodel has its specific tool to be implemented. This result intersects with the *Executable DSML pattern* that favors the definition of generative tools and thus eases the integration of tools for new DSMLs.

5.3.2 Architecture of FEVEREL

FEVEREL is a domain-specific language proposed for a DSML designer to manage verification results. It allows to define how a DSML event can be observed at the formal level. The architecture of FEVEREL is shown in Figure 5.9. The entry-point is a FEVEREL model defined by the DSML designer. A FEVEREL editor serves as an interface to ease the DSML designer task. According to the Figure 5.1, we decided to automatically generate the second transformation which manage verification results from a FEVEREL model.

Based on the DSML metamodel, the formal language metamodel and theirs semantic metamodels, the DSML designer defines a mapping between DSML events defined in the EDMM of the DSML and their corresponding elements in the formal side (FEVEREL *model* in Figure 5.9).

For example, FEVEREL allows to specify that the SPEM *StartWD* event of a wokdefi-

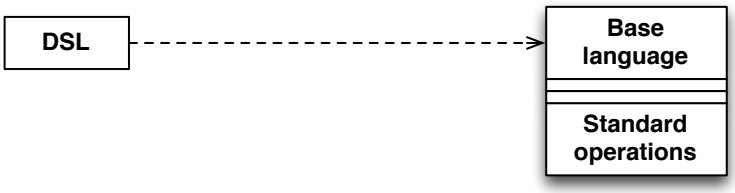

Figure 5.10 — The piggyback pattern

nition *wd* can be observed when there is a *FireTransitionEvent* instance in the TPN scenario whose name is the concatenation of *wd* with '_start'.

From a FEVEREL model, a model to model transformation (*FormalScenario2DSMLScenario* in Figure 5.9) is automatically generated. This transformation translates verification results (*Formal scenario* in Figure 5.9) generated by model checking tools into a DSML scenario easier to understand by the DSML end-user.

Due to this architecture, the DSML designer obtains a suitable tool to define mapping between events. He does not deal with technical aspects of a model transformation. The DSML designer aims to describe how a DSML behavioral element can be observed in the formal side relying on behavioral extensions of both sides and the defined translational semantics which maps the abstract syntax of the DSML into the formal ones.

5.3.3 Implementation of FEVEREL language

A DSL (whether it is a DSML or a DSPL) design should be suitable for the user and especially corresponds to his abilities in software engineering. In many cases, designing a new DSL is one of the core challenges of modern software engineering as it is an error-prone and time consuming task [KKP+09].

So, it is mandatory to adopt strategies to design such DSLs in order to ease the definition of the abstract and concrete syntaxes and the semantics. In [Spi01], a study was developed which introduces eight DSL design patterns. To develop the FEVEREL DSPL, we choose to implement two patterns: the **piggyback pattern** and the **source to source transformation pattern**.

The piggyback pattern (Figure 5.10) proposes the use of an existing language as a hosting base for the new DSL. This hosting language can be a general-purpose language which offers to the DSL standardization and powerfulness and makes it more user-friendly. The DSL can, therefore, share common syntactical elements such as expression handling, operations, arithmetic and logic operators, etc.

The source to source transformation pattern (Figure 5.11) allows to implement efficiently a DSL translator. It can be used to ease the implementation of a DSL and to help the DSL designer by leveraging the facilities provided by existing language tools. The DSL code can be translated using an appropriate translation process into the source code of the existing language. Available tools for the existing language are then used to host the generated code.

Figure 5.11 — The source to source transformation pattern

For the DSML designer, the expected tool should be a user-friendly tool, close in syntax and semantics to his skills and correspond to his casual capabilities like metamodeling techniques using ECORE, expressing constraint languages with OCL.

Figure 5.12 shows the implementation of the FEVEREL language. It implements the piggyback pattern with OCL as base language (the blue dashed arrow).

The second part of the implementation of FEVEREL language concerns the translation part (the red dashed arrows).

The source to source transformation pattern has been used to ease the burden of implementation. We have chosen the ATL transformation language as a host language (ATL.ecore) because we aim to automatically generate an ATL model transformation from a FEVEREL model. Source to source transformation pattern intersects with an interesting MDE technique which is higher-order transformations technique. Therefore, the DSPL compilation is considered as a higher-order model transformation (FeVeReL2ATL.atl).

Combining the piggyback pattern with the source to source transformation pattern shows two advantages. First, a big part of the translation is the identity. As we use (1) OCL as a base language to implement our DSML and (2) the ATL which is based on OCL as a host language to apply source to source transformation pattern, the translation is focused on the domain-specific elements. Second, often DSMLs evolve and can be extended. So, an eventual extension of FEVEREL will be easily adopted by a DSML designer because he only needs to update the semantics with new domain-specific elements by extending the higher-order transformation while the OCL part is unchanged.

5.3.4 Syntaxes and semantics of FEVEREL

In this subsection, we will detail the elements of our implemented DSPL. We illustrate it with the XSPEM case-study. Let's consider the translational semantics proposed in the chapter 3 and the two applications of the *Executable DSML pattern* on the SPEM metamodel (Figure 3.7) and the TPN metamodel (Figure 3.8).

Listing 5.5 shows the use of FEVEREL in this case-study. First, it is mandatory to declare different DSML metamodels (lines 2-3). *DSMLMetamodel* represents the classical metamodel of the DSML. *DSMLSemantics* extends the first metamodel with the application of the *Executable DSML pattern* on the DSML. In addition, we declare formal language metamodels. *FormalLanguageMetamodel* shows the abstract syntax of the formal metamodel, and the *For-*

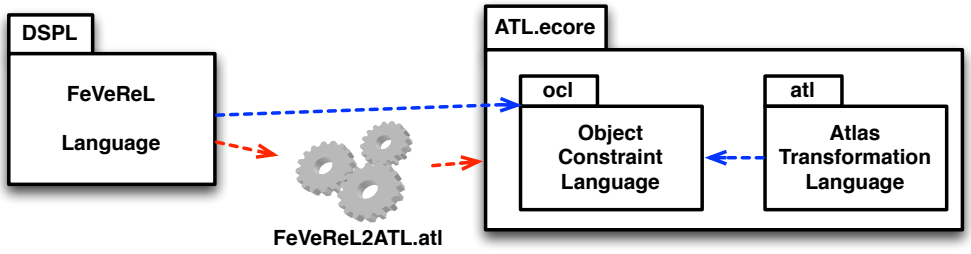

Figure 5.12 — Implementation of FeVeReL DSPL

malLanguageSemantics shows the behavioral extension done on the initial metamodel.

Now, we are ready to explicit different mappings between events. The first mapping named *StartWD_As_Transition* (lines 10-21) (respectively *FinishWD_As_Transition* (lines 24-35)) shows how a `StartWD` (a `FinishWD`) event of a workdefinition in the DSML level can be observed in the formal level.

In fact, this event corresponds to an instance of the `FireTransitionEvent` meta-class, captured in the formal scenario, whose name of its fired transition is the concatenation of the concerned workdefinition name to `'_start'` (line 19) (respectively `'_finish'` (line 23)).

```
1  // DSML metamodels declaration
2  import "http://SPEM/1.0" as DSMLMetamodel
3  import "http://SPEMSemantics/1.0" as DSMLSemantics
4
5  // Formal language metamodels declaration
6  import "http://TPN" as FormalLanguageMetamodel
7  import "http://TPNSemantics" as FormalLanguageSemantics
8
9  // The mapping between the StartWD event and the corresponding one in the TPN level
10 events mapping StartWD_As_Transition:
11
12         DSMLEvent swd_event: DSMLSemantics.StartWD(
13                 date <- tpn_event.date
14         )
15
16         maps
17
18         FormalEvent tpn_event : FormalLanguageSemantics.FireTransitionEvent(
19                         tpn_event.firedTransition.name = swd_event.workdefinition.name+'_start'
20         )
21 end events mapping
22
23 // The mapping between the FinishWD event and the corresponding one in the TPN level
24 events mapping  FinishWD_As_Transition:
25
26         DSMLEvent fwd_event: DSMLSemantics.FinishWD (
27                 date <- tpn_event.date
28         )
29
30         maps
31
32         FormalEvent tpn_event: FormalLanguageSemantics.FireTransitionEvent (
33                 tpn_event.firedTransition.name = fwd_event.workdefinition.name+'_finish'
34                 )
35 end events mapping
```

Listing 5.5 — The definition of events mappings using FeVeReL in the case-study of the verification of SPEM models using TPN

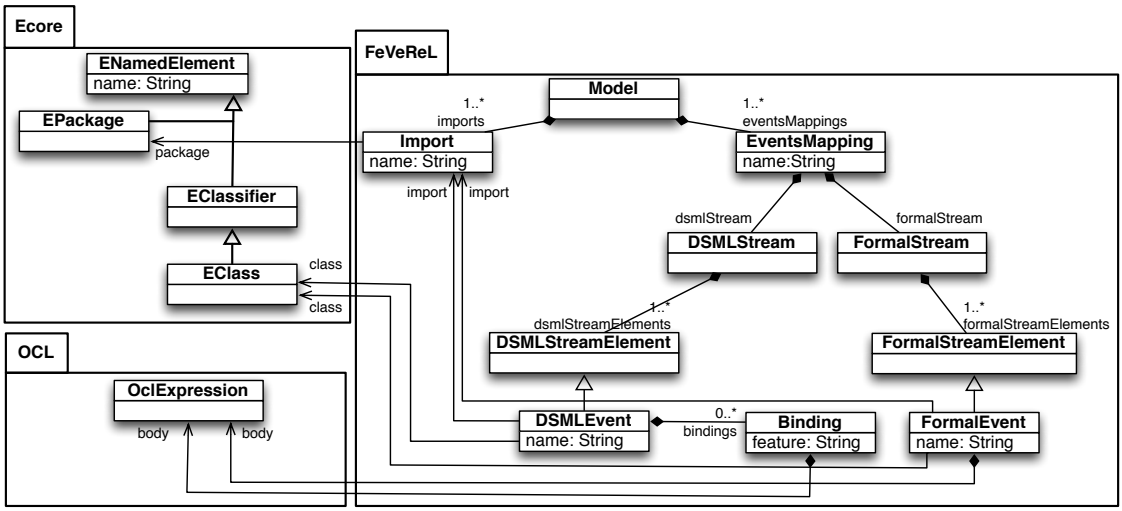

Figure 5.13 — The metamodel of FEVEREL DSPL

The use of the FEVEREL language offers for the DSML designer a more structured tool to specify how verification results should be brought back to the DSML level. Based on the defined translational semantics and both extensions, he can easily define a kind of mapping based on OCL to identify which formal event corresponds to a DSML one.

Let's detail different FEVEREL elements:

Syntax The FEVEREL syntax is defined using Xtext. A subset of the FEVEREL metamodel is shown in Figure 5.13. A *Model* is composed of a set of *imports* for the concerned metamodels. In addition, it contains a set of mappings between events (*eventsMappings*). An *eventsMapping* is characterized by an identifier. It describes a mapping between an observation events kind in the DSML side (*DSMLStream*) and its corresponding one in the formal level (*FormalStream)*.

A *DSMLStream* contains a set of elements (*DSMLStreamElement*) which allow to structure a possible observation of events. Currently, a *DSMLStreamElement* is only one event. Extensions to more events are part of future work. A *DSMLEvent* is characterized by an identifier. It refers to a meta-class in the metamodel of the DSML semantic extension. In addition, it defines a set of bindings (*Binding*) which specify the initialization of a feature (an attribute or a reference) of a DSML event using an expression (*body*).

In the formal side, as in the DSML side, a *FormalStream* contains only one event. A *FormalEvent* has an identifier. It refers to a meta-class of a formal event in the formal semantic extension. It contains also an OCL expression (*body*). It could also contain several events thus allowing n-to-m mapping for language whose semantics are not structurally aligned.

Semantics As shown in the subsection 5.3.3, FEVEREL has a translational semantics. We choose to map the FEVEREL abstract syntax into the well-known model transformation language ATL. Let's detail the main transformation rules:

- Each *EventsMapping* is translated into:

    1. an ATL helper. Its context is the super type of all kind of events in the EDMM extension of the formal metamodel. It does not contain parameters. The returned type of this helper corresponds to a set of elements in the DSML DDMM on which the *DSMLEvent* is instantiated. The body of this helper is shown as a selection of a subset from all instances of this element in the DSML DDMM for which the body of the *FormalEvent* evaluates to true.
    2. a lazy rule. Its source pattern is the element in the DSML DDMM on which the *DSMLEvent* is instantiated and its different features (*features*) defined in the *DSMLEvent*. The target pattern creates an instance of the *DSMLEvent* with different features declared in the source pattern.

- The FEVEREL *Model* is translated into ATL rules. Its source pattern is a formal scenario (an instance of the *Scenario* meta-class in the TM3 of the formal metamodel). Its target pattern is a DSML scenario (an instance of the *Scenario* meta-class in the TM3 of the DSML metamodel). It aims to produce a DSML scenario from the formal one. The formal scenario is iterated using an `iterate` expression. The iterated variable which is the current instance of the *FormalEvent* will check, by calling the helper generated previously, whether there are elements in the DSML DDMM which satisfy the body of the helper. According to this result, the lazy rule will be called for each element of the returned subset and with its corresponding features.

This translation, `FeVeReL2ATL` is implemented using ATL which has several facilities to implement higher-order transformations.

This approach follows the previous one experimented to define the behavioral properties on the DSML level and generate formal ones in the formal level. The most important point is that the DSML designer will be guided to generate a kind of "DSML verification framework" for each DSML. It is a structured approach to generate this framework based on the *Executable DSML pattern*.

5.4 Related works

The problem of integrating formal verification into the design of DSMLs has been widely addressed by the MDE community. However, the analysis feedback at the DSML level problem is typically either ignored or resolved by defining ad-hoc or hard-coded solutions. For example, in [OO+12], authors propose an approach, named *Metaviz*, based on the real-time systems specification and validation tool set IFx-OMEGA. It is designed to ease the visualization of the simulation trace. The goal is to assist the user in the Interactive Simulation task by refining this step with a diagnosis process built around visualization concepts. It consists in feeding back verification results at OMEGA level. Thus, It can be considered as an ad-hoc approach. An eventual application of the *Executable DSML pattern* on both domains can ease the integration of our FEVEREL language in their approach. On the other hand, a few number of works handling the feedback with general solutions exists in the literature.

In [CGR11], authors introduce an algorithm requiring the DSML's semantics to be defined formally, and a relation R to be defined between states of the DSML and states of the target language.

The DSML designer must provide as input a natural-number bound n, which estimates a difference of granularity between the semantics of the DSML and the semantics of the target language. However, we don't think that DSML designer, for who it is difficult to use formal methods and verification, can define this important information to feedback verification results.

Hegedüs et al. [HBRV10] propose a technique for the back propagation of the simulation traces based on change-driven model transformations from traces generated by SAL model checking framework to the specific animator named BPEL Animation Controller.

So, they define a change-driven model transformation which consumes changes of the Petri nets simulation run and produces a BPEL process execution using traceability information generated while running the translational semantics defined previously. In this case, after defining the runtime extension for both levels (BPEL and Petri nets) and the translational semantics, the DSML designer is invited to define 1) a change command metamodel for Petri nets and BPEL and also 2) the backward change-driven transformation. In our approach, we try to give for the DSML designer a high-level tool to define mapping between events.

In [GdLMD09], a domain-specific visual language called *BaVeL* is designed. It allows defining how a verification result soulhd be reflected in terms of the original notation. It is based on triple graphical patterns.

This approach requires an additional information which is the mappings (named also traces) relating the source and target models and created during the running the translational semantics. This framework could even be implemented using the QVT model transformation language as it creates traces between the source and target models. Usually, DSML designers choose to encode the translational semantics as a code generation process (model-to-text transformation) instead of a model-to-model transformation. So, this information is missed. In our approach this information is optional but not an essential one. The DSML designer decides if it is required to generate model transformation traces to ease the feedback with FEVEREL. He must import the mappings metamodel in his FEVEREL specification.

6 Building a verification framework for an executable DSML

Résumé

Les deux chapitres précédents ont présenté nos contributions pour étendre la chaîne outillée de vérification pour un nouveau DSML avec les éléments nécessaires pour exprimer des propriétés comportementales au niveau du DSML et pour générer les propriétés formelles correspondantes; puis pour remonter les résultats de vérification du niveau formel vers le niveau DSML. Nous avons ainsi une approche pour définir une sémantique d'exécution pour les DSMLs et l'outillage nécessaire pour obtenir une chaîne outillée complète qui enrichit un DSML avec les capacités de V&V.

Pour faciliter le développement d'un framework de vérification pour des nouveaux DSMLs, nous avons suivi une approche dirigée par des exemples pour obtenir à la fin les outils appropriés pour les experts d'un DSML, le concepteur d'un DSML et les utilisateurs finaux d'un DSML. Au début, nous avons écrit les propriétés comportementales attendues au niveau formel manuellement et nous les avons testées avec la boîte à outils TINA. Ensuite, pour les générer, nous avons commencé par écrire manuellement la transformation modèle à texte. Cette solution n'aide pas l'expert du DSML dans la spécification et la mise en œuvre de ses propriétés comportementales. Il s'attend à obtenir un langage plus approprié pour les spécifier. Par conséquent, nous avons identifié les différents éléments qui peuvent être capitalisés. Le QDMM a donc été identifié. En outre, nous avons défini une extension temporelle du langage OCL, appelée TOCL, puis nous avons spécifié comment les différentes constructions de TOCL doivent être transformées en ATL afin de générer la transformation modèle à texte.

Pour faciliter la remontée des résultats de vérification pour le concepteur du DSML, nous avons commencé par la spécification de plusieurs couples de modèles conformes à un DSML et les scénarios attendus présentés sous forme d'une succession ordonnée d'événements du DSML (des instances de l'EDMM du DSML). Ensuite, nous avons écrit manuellement la transformation de retour pour engendrer ces scénarios automatiquement. Si cette solution est fonctionnelle, elle oblige les utilisateurs à savoir écrire les transformations de retour nécessaires. Il semble plus judicieux de fournir un langage qui permet à l'utilisateur de spécifier le retour en s'appuyant sur les extensions apportées par le patron de métamodélisation. Les transformations sont alors engendrées automatiquement de cette description.

Ce chapitre est organisé comme suit. La section 6.1 montre un aperçu complet d'un framework de vérification pour un DSML. La section 6.2 introduit le processus requis pour générer un framework de vérification pour un nouveau DSML et souligne les interactions nécessaires entre les différents acteurs. La section 6.3 explique les dépendances entre les différents éléments d'un framework généré. La section 6.4 montre la validité de notre approche. Enfin, nous concluons à la section 6.5.

BOTH previous chapters introduced our contributions to extend the verification toolchain for a new DSML by the elements to express user level behavioral properties and generate the corresponding formal ones; and then to feedback verification results from the formal level to the DSML user one. We proposed an approach to introduce the executability aspect for DSMLs and the required tooling to obtain a complete toolchain which empowers a DSML with V&V capabilities [ZCP13a, ZCP14].

To ease the development of a verification framework for a new DSMLs, we followed an example-driven approach to obtain at the end the appropriate tools for the DSML experts, DSML designer and DSML end-users. At first, we wrote the expected behavioral properties manually and we tested them with the TINA toolbox. Then, to generate them, we started by writing manually the model-to-text transformation. This solution does not help the DSML expert in specifying and implementing his behavioral properties. He expects obtaining a more suitable language to specify them. Consequently, we identified different elements that can be abstracted and capitalized. The QDMM was thus identified. In addition, we defined a temporal extension of OCL and then we specified how different TOCL constructs should be translated to ATL in order to generate the model-to-text transformation.

To ease the feedback of verification results for the DSML designer, we started by specifying several pairs of DSML conforming models and the expected scenarios shown as an ordered set of DSML events (instances of the DSML EDMM). Then, we wrote manually the backward transformation to generate these scenarios. This solution is repetitive and does not seem to coincide with users needs. They aim to obtain a small language to specify how this feedback should be done based on runtime extensions previously specified.

In this chapter, we propose a method to build such a tool chain and we explain the interactions with the concerned actors, the DSML expert, the Formal Methods expert and the DSML designer. It is appropriate to give for these concerned actors a complete overview of our approach and how to use it to build a verification framework for a new DSML.

This chapter is organized as follows. Section 6.1 shows a complete overview of a DSML verification framework. Section 6.2 introduces the required process to generate a verification framework for a new DSML and stress the required interactions between different actors. Section 6.3 explains the dependencies between different elements of the generated framework. Section 6.4 shows the validity of our approach. Finally, we conclude in section 6.5.

6.1 Architecture of the verification framework for a new DSML

Figure 6.1 shows a complete overview of a verification framework for a new DSML with its different ingredients.

Blue boxes show the different required elements that the DSML designer should implement to generate a verification framework for a new DSML. Yellow boxes show the different generic elements provided for him to translate his specifications into the appropriate format. These ones may rely on generated elements (the green boxes). The DSML verification framework can be subdivided into three levels.

The first level concerns the mapping of DSML conforming models (*myModel.dsml*) into

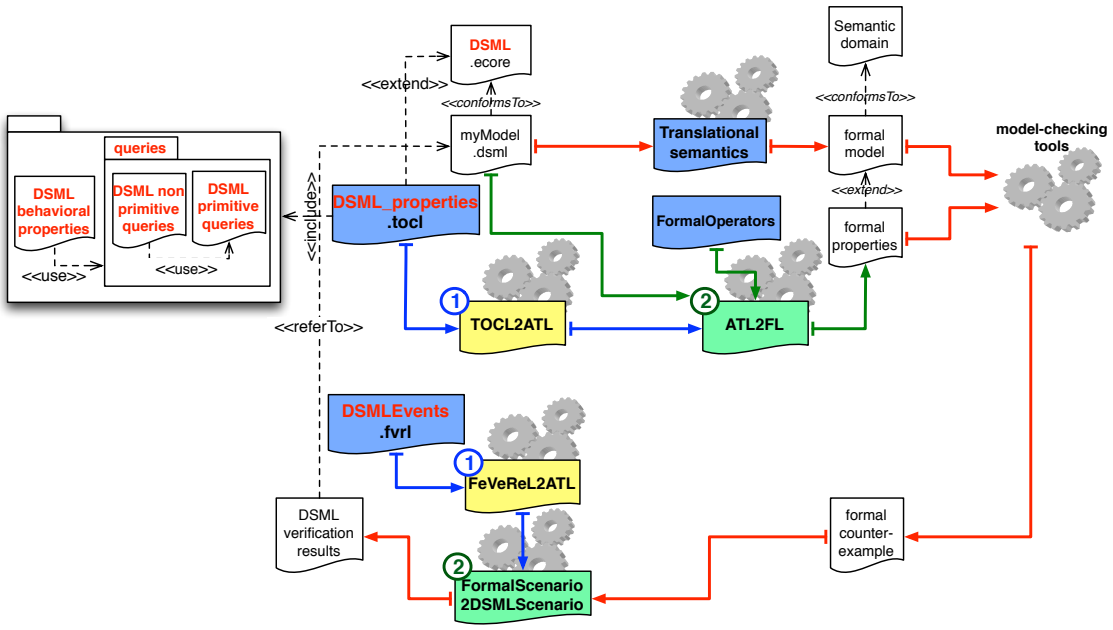

Figure 6.1 — An overview of a generated verification framework for a new DSML

the semantic domain (*formal model*). It is implemented by the *translational semantics* to generate a formal model in the input format of model-checking tools.

The second level concerns the expression of DSML behavioral properties and the generation of the corresponding formal ones. It uses the TOCL editor to define DSML queries and their related behavioral properties (DSML *_properties.tocl*). We provide the mandatory generic tooling (TOCL2ATL) to generate a DSML-specific query (ATL2FL) which takes a DSML conforming model and generates the corresponding formal properties. Before that, once the formal language is chosen, it is required to define the library of formal operators (*FormalOperators*) to ease the generation of formal properties.

The third level concerns the feedback of verification results. The DSML designer provides mappings to explain how DSML events can be observed in the formal side. We provide a generic HOT transformation (FeVeReL2ATL) that generates a model-to-model transformation that transforms formal counter-example into DSML verification results.

6.2 The generation of a verification framework for a new DSML

In this section, we focus on the generation of a verification framework for a new DSML. First, we identify the different actors who participate in this process. Then, we explain the complete process to generate the framework and the different interactions between concerned actors.

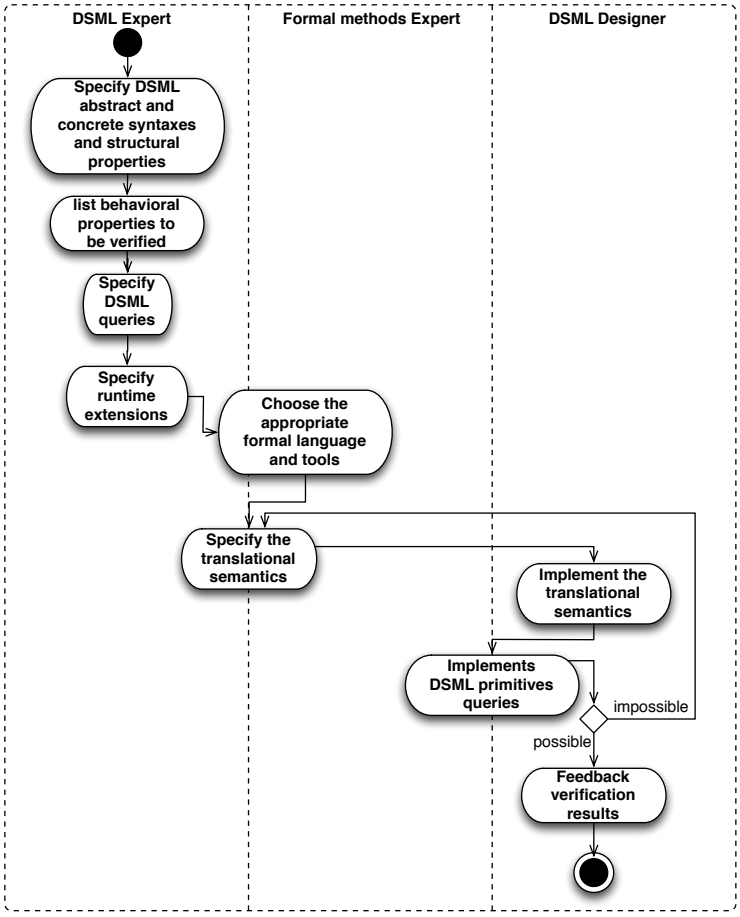

Figure 6.2 — DSML verification framework generation process

6.2.1 Identification of different actors

Let's identify different actors who are involved in the development of a DSML verification framework.

6.2.2 The process of DSML verification framework generation

The **DSML Expert** is the domain specialist. He defines the DSML which meets the DSML end-user needs. He must create a metamodel for a new language and the tooling (structural properties, concrete syntaxes, editors). Here we focus on the verification activity. The DSML expert identifies the kind of behavioral properties (safety and liveness) that should be verified on models. In doing so, he specifies the queries that are needed to express these DSML properties. These queries are expressed in the QDMM. The **Formal Methods Expert** is a specialist in formal methods. He has technical and theoretical skills on several techniques and tools used to perform verification activity. He defines the outline of the translational semantics based on the DSML expert needs.

The **DSML Designer** is a software language specialist. He has capabilities in the MDE and formal methods domains. He implements verification activity for a new DSML. First,

he implements the translational semantics according to the specification proposed by the Formal methods expert and validated by the DSML expert. Then, he implements different DSML extensions to build the DSML verification framework.

The main steps that are part of the DSML verification framework generation are shown in Figure 6.2 and explained here after. It shows the organisational process to generate the DSML verification framework and interactions between concerned actors.

The starting point of the process is performed by the DSML expert. He defines the abstract and concrete syntaxes of the DSML. In addition, he implements structural properties related to the DSML conforming models. This step focus only on structural aspects of the DSML. It defines the entry point to apply our contributions which aim to add the possibility to verify formally and automatically behavioral properties while hiding all formal aspects for the DSML end-users.

Now, the DSML expert can start the process of integration of the behavioral verification tool for his DSML. At first, he must specify his needs by identifying informally the behavioral properties to verify on DSML conforming models. Then, he can formalize these properties using the TOCL editor. This step allows to identify and specify the different queries that are defined in the QDMM. This information helps the DSML end-user to assess the model during execution. Thereafter, the DSML expert defines such pairs of models that do not verify his properties and also the expected verification results which are shown as a scenario containing a set of DSML events triggered and DSML states observed during the execution. Thereby, the EDMM and SDMM runtime extensions are defined.

Next, the Formal Methods expert and the DSML expert collaborate to choose the appropriate formal language and tool to specify the translational semantics for this DSML based on the expected behavioral properties. Then, they specify together the translational semantics.

Therefore, the DSML designer implements the specified translational semantics which provides the observers required by the identified queries. It consists in mapping the abstract syntax elements of the DSML into the chosen formal language. Furthermore, he must implement the previously identified primitive queries which specify observers that should be available in the formal model. In addition, the DSML designer must define different behavioral extensions (EDMM and SDMM) on the formal level. He extends them with the necessary tools to transform the model-checker results into a formal scenario. These tools are defined at once but reused for other DSML toolchains. The last step concerns the feedback of verification results. The DSML designer should define a mapping between the formal events and their corresponding ones in the DSML level. The DSML verification framework is defined when all these steps are performed.

6.3 Dependencies between DSML verification framework elements

Once the verification framework has been developed, several modifications can be proposed to improve it like verifying new kinds of properties, capturing other kinds of events or

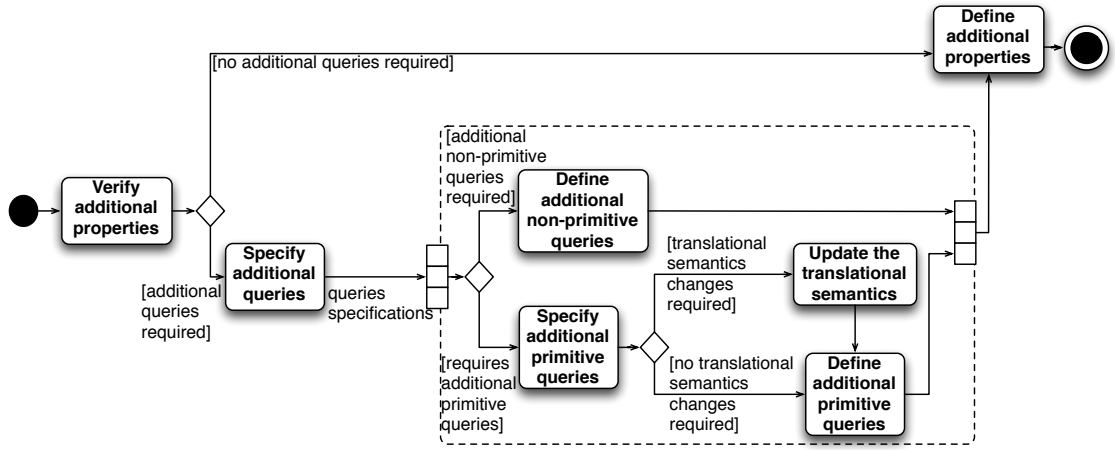

Figure 6.3 — Additional behavioral properties verification process

queries, reusing another simulator or formal tools, etc. In the following, we provide a non exhaustive list of possible modifications on the DSML verification framework and identify their impacts on its different related DSML verification framework elements:

1. The DSML expert or the DSML end-user needs to **assess additional behavioral properties**: it is required to check whether these ones require defining additional queries. If the existing queries are not enough to define these properties, then, additional queries should be specified. These queries can be primitive or non-primitive ones. The definition of primitive queries requires checking whether the translational semantics should be updated. Figure 6.3 shows the activity diagram illustrating this process.

2. An eventual **evolution in the translational semantics** to target additional properties might be needed: the primitive queries and the events mappings can be affected by this change. If this is the case, it is mandatory to update their definitions. For example, in the translational semantics from SPEM to TPN shown in chapter 3, if the DSML designer chooses to encode the *finished* place differently, he should also update the XSPEM related primitive query *isFinished()*. Figure 6.4 shows the translational semantics change process.

3. If the DSML designer chooses to **change the target formal language**, it is mandatory to update the *formalOperators* library in order to correctly generate behavioral properties in the formal side because the formal operators symbols change between different model-checking tools. For example, the binary temporal operator **implies** is coded in TINA as `=>`, but in Promela language[1], the input format of SPIN model-checker [Hol03], it is defined as `->`. In addition, the runtime extensions (EDMM and SDMM) and the required tools to generate a formal scenario must be updated in the formal side. Finally, it is necessary to update the translational semantics.

4. A possible **evolution on the model-checker or the use of another one**. This change

[1] `http://spinroot.com/spin/Man/Intro.html`

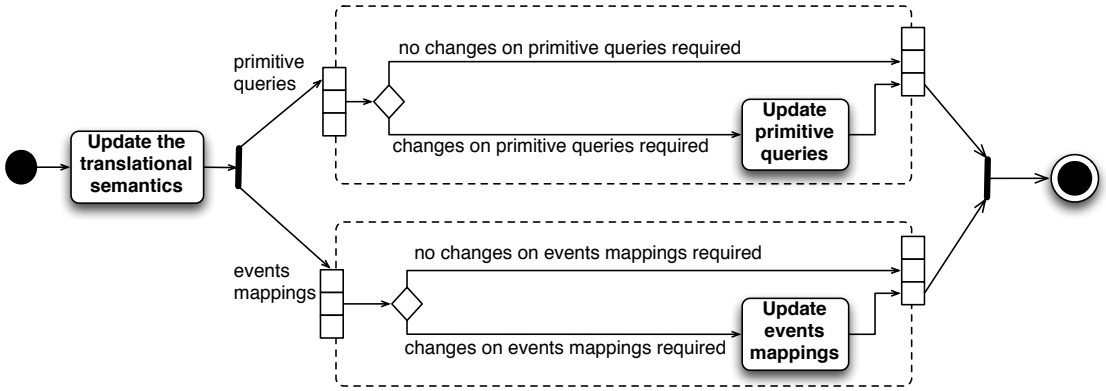

Figure 6.4 — Update the translational semantics process

requires only adapting the provided tooling defined by the DSML designer to generate the scenario on the formal side.

To summarize these relations, Figure 6.5 shows different dependency relations between different verification framework elements.

6.4 Guidelines for validating the verification toolchain

Defining a translational semantics is a highly creative activity which requires high skills both in the formal language and in the DSML to find an efficient mapping between both languages as well as in transformation techniques. We thus only provide guidelines to favor the definition of a correct transformation.

A first guideline is the obligation to define for each QDMM primitive query the corresponding formal property language (like LTL) fragment. QDMM queries are thus a kind of checklist that ensures that all aspects of interest for the DSML end-user have indeed been modeled on the formal side.

A second guideline to validate the translational semantics consists in formalizing invariants on the DSML using TOCL and then automatically translating them on the formal side. These ones assert semantics properties of the DSML that must be preserved by the encoding. There might be pre-post conditions for each EDMM and a protocol state machine for the EDMM. If they fail, an error is detected (either in the translation, the invariants or the queries implementations). Let's illustrate this with our XSPEM translational semantics into TPN.

Listing 6.1 shows the possible formalization of some invariants using our TOCL editor. The first *mutex* invariant (lines 9-10) checks whether different workdefinition's states are mutually exclusive. We define additional primitive queries (*isRunning()* and *isReady()*[2]) to enable to capture different XSPEM workdefinition's states and thus to formalize the *mutex* invariant.

[2]It corresponds to the *_isNotStarted* place in the TPN model

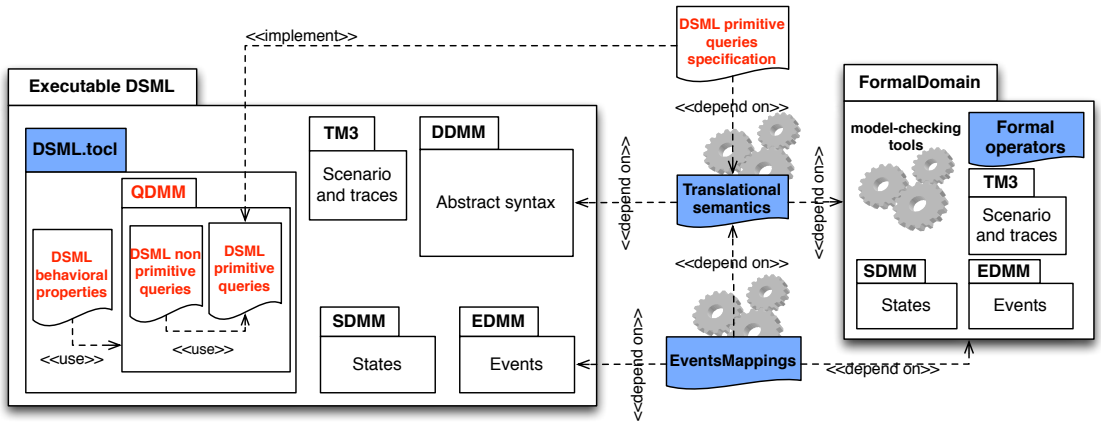

Figure 6.5 — A Dependencies view of the generation of behavioral properties

```
1  // Additional primitive queries
2  context SPEM!WorkDefinition def : isRunning(): String=
3          self.name+'_running';
4
5  context SPEM!WorkDefinition def : isReady(): String=
6          self.name+'_notStarted';
7
8  // Translational semantics validation
9  context SPEM!WorkDefinition inv mutex:
10         always (self.isReady() or self.isRunning() or self.isFinished())
11
12 context SPEM!WorkDefinition inv  precedence_same_workdefinition:
13         self.isStarted() before self.isFinished()
14
15 context SPEM!WorkSequence inv precedence_between_different_workdefitions:
16         if (self.linkType = #startToStart)
17         then
18                 self.predecessor.isStarted() before self.successor.isStarted()
19         else
20                 if (self.linkType = #startToFinish)
21                 then
22                         self.predecessor.isStarted() before self.successor.isFinished()
23                 else
24                         if (self.linkType = #finishToStart)
25                         then
26                                 self.predecessor.isFinished() before self.successor.isStarted()
27                         else
28                                 self.predecessor.isFinished() before self.successor.isFinished()
29                         endif
30                 endif
31         endif
```

Listing 6.1 — Validation of the translational semantics of xSPEM into TPN

The second validation invariant, *precedence_same_workdefinition* (lines 12-13) checks whether the execution semantics of each xSPEM workdefinition is preserved. It means that each workdefinition can finish only if it was started. The last validation invariant, *precedence_between_workdefitions* (lines 15-31) checks whether the different dependency constraints expressed via the *linkType* attribute of *WorkSequence*, are preserved during the execution. For example, if the *linkType* of a worksequence is *startToFinish* (line 22), the workdef-

inition predecessor should be started before that the workdefiniton successor finishes.

The validation invariants must be defined by the DSML expert to help the DSML designer in implementing the translational semantics. In addition, the DSML must complete the required additional queries primitive queries.

This additional specification can be translated using our TOCL tooling to generate their corresponding LTL ones that must be assessed on TPN models generated from XSPEM conforming models.

6.5 Conclusion

We have presented an user-oriented approach to integrate behavioral verification tools on a new DSML in order to assist the DSML designer into the building of a verification framework for a new DSML. This framework allows the DSML end-user to verify safety and liveness properties on executable models. We give the required steps to generate this kind of framework and what are the different need exchanges between concerned actors. This framework should hide different formal methods particularities to the DSML end-user who masters only his domain notions. Then, we present a complete overview of such a verification framework for a new DSML. We show dependencies between the different elements of the verification framework in order to explicit the actions the DSML designer must undertake when some evolutions are required. Finally, we have provided some elements to validate the verification toolchain.

Part

Validation of the approach

7 Application of the approach using an intermediate language

Résumé

Nous avons proposé dans la partie précédente une approche générique pour intégrer l'activité de vérification à un nouveau DSML. Cependant, cette approche s'appuie sur la définition d'une sémantique translationnelle qui peut être complexe quand il y a un écart sémantique important entre le DSML et le domaine formel.

Pour réduire cet écart, un langage intermédiaire peut être introduit dans la chaîne outillée de vérification. Il fournit un haut niveau d'abstraction pour les formalismes de vérification utilisés dans les différentes boîtes à outils. Les transformations entre ces langages et les outils formels sont définies une seule fois et sont partagées par tous les DSMLs. FIACRE [BBF+08] est un exemple d'un tel langage intermédiaire. C'est un langage de spécification formelle qui vise à la fois les aspects comportementaux et de synchronisation des systèmes temps réel. Il a été conçu comme langage cible dans le projet TOPCASED pour les transformations de modèles de différents DSMLs tels que AADL, BPEL ou SDL (Figure 7.1).

L'intégration s'appuie sur notre étude méthodologique proposée au chapitre 6. Elle consiste à substituer le domaine formel par un autre, plus proche de la sémantique des DSMLs.

Dans ce chapitre, on introduit FIACRE comme un langage intermédiaire dans une chaîne de vérification afin de réduire l'écart sémantique entre les DSMLs et les langages formels grâce aux constructions de haut niveau de FIACRE. Ceci illustre également la généricité de notre approche puisqu'on substitue le langage formel cible initial, TPN, par un autre, FIACRE. La première section introduit le langage FIACRE et l'illustre avec une implémentation de l'algorithme de Peterson d'exclusion mutuelle. En outre, on montre ce qu'offre FIACRE pour spécifier et implémenter des propriétés comportementales.

La section 7.3 montre les différentes extensions comportementales (EDMM et SDMM) de FIACRE introduites par l'application du patron de métamodélisation. Ces extensions sont indispensables pour l'intégration de FIACRE dans la chaîne de vérification. La section 7.4 décrit comment intégrer FIACRE d'une manière transparente dans notre approche pour construire une chaîne outillée de vérification. On explique les étapes requises pour remonter les résultats de vérification depuis les langages formels de bas niveau vers le langage intermédiaire FIACRE.

Finalement, en s'appuyant sur notre étude méthodologique (chapitre 6), la section 7.5

montre les étapes nécessaires d'évolution quand on choisit de substituer le langage formel TPN par FIACRE dans la chaîne de vérification des modèles SPEM.

WE have proposed a generic approach to integrate the verification activity for a new DSML in the previous part. However, this approach relies on the definition of the translational semantics which may be complex when there is a big semantic gap between the DSML and the formal domain.

To bridge this gap, an intermediate language can be integrated in the verification toolchain. It provides a high-level of abstraction of the verification formalism used in the various toolsets. Transformations between it and formal tools are the defined once and shared for all DSMLs. FIACRE [BBF+08] is an example of such an intermediate language. It is a formal specification language that targets both the behavioral and timing aspects of real-time systems. It was designed as the target language in the TOPCASED project for model transformations from different DSMLs such as AADL, BPEL or SDL (Figure 7.1).

This integration is based on our methodological study proposed in chapter 6. It consists in substituting the formal domain by another one closer to DSMLs semantics.

In this chapter, we introduce FIACRE as an intermediate language in the verification chain in order to reduce the semantic gap between DSMLs and formal languages thanks to the high level constructs which are part of FIACRE. This also illustrates the genericity of our approach as we substitute the initial target formal language, TPN, by another one, FIACRE. The first section introduces the FIACRE language and illustrates it with an implementation of the Peterson's exclusion algorithm. In addition, we show how we can specify behavioral properties at the FIACRE level.

Section 7.3 shows different behavioral extensions (EDMM and SDMM) done on FIACRE by applying the *Executable DSML pattern* in order to integrate it in the verification toolchain. Section 7.4 describes how we can transparently integrate FIACRE in our approach to build a verification toolchain. We explain the required steps to feedback verification from low-level formal languages to the FIACRE intermediate language.

Finally, based on our methodological contribution explained in the chapter 6, the section 7.5 shows the required steps when we choose to substitute the TPN by FIACRE for the verification of SPEM models.

7.1 The Fiacre Language

FIACRE is a french acronym for an Intermediate Format for Embedded Distributed Components Architecture. It was designed as the target language for model transformations from different DSMLs such as AADL [CBF+10], PLC [FDQDR+11] or SDL [CBG+08] targeting different verification languages and toolset like TINA and CADP.

FIACRE is a formal language to represent both the behavioral and timing aspects of systems, in particular embedded and distributed systems, in formal verification and simulation purposes. FIACRE is built around two notions:

- Processes describe the behavior of sequential components. A process is defined by a set of control states, each associated with a piece of program. FIACRE contains different deterministic constructs available in classical programming languages like assignments,

if-then-else conditionals, while loops, and sequential compositions. In addition, it provides two non deterministic constructs: non deterministic choice (`select` operator) and non deterministic assignments. Communication between processes is ensured via ports, and jumping to a next state is done with the `to` or `loop` operators.

- Components describe the composition of processes, possibly in a hierarchical manner. A component is defined as a parallel composition of instantiated components and/or processes communicating through ports and shared variables. The notion of component also allows first to restrict the access mode and visibility of shared variables and ports, then to associate timing constraints with communications, and last to define priority between communication events.

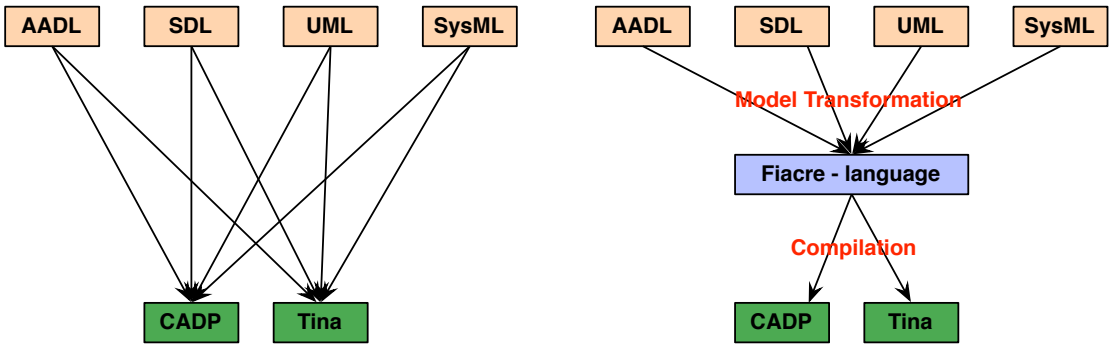

Figure 7.1 — FIACRE as intermediate language to reduce complexity when targeting several formal toolboxes from modeling languages

We give in Listing 7.1 an implementation of Peterson's algorithm [Pet81] with FIACRE. It is a concurrent programming algorithm for the mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication.

The algorithm uses two variables, *flag* and *turn*. A *true* value of *flag* indicates that the process wants to enter in the critical section. The variable *turn* holds the identifier (0 for P_0 and 1 for P_1) of the process whose is allowed to access.

To implement this algorithm with FIACRE, a type declaration *id* is defined (line 3). It allows to identify different processes. In addition, a type declaration *flag* is defined (line 5) in order to declare a boolean array of size 2.

```
1  /* Types */
2
3  type id is 0..1
4
5  type flag is array 2 of bool
6
7  /* Processes */
8
9  process Proc(pid : id, &flag : flag, &turn : id) is
10   states idle, waits, CS
11   from idle
```

```
12        flag[pid] := true ;
13        turn := 1 - pid ;
14        to waits
15
16     from waits
17        on not (flag[1 - pid] and turn = 1 - pid);
18        to CS
19
20     from CS
21        /* do something in the critical section */
22        flag[pid] := false ;
23        to idle
24
25 /* Main component */
26
27 component Main is
28     var flag : flag := [false, false] ,
29         turn : id := 0
30
31     par
32        Proc (0, &flag, &turn)
33     || Proc (1, &flag, &turn)
34     end
35
36 /* Entry point for verification */
37
38 Main
```

Listing 7.1 — An implementation of Peterson's algorithm with FIACRE

Next, a *Proc* process is defined. It has three parameters: its identifier (*pid*), the *flag* shared array and the *turn* shared variable. In addition, it contains three states *idle*, *waits* and *CS* (to specify the critical section) (line 10).

Three transitions are declared to explain the behaviour of a process. The first one (lines 11-14) indicates that the process wants to enter in the critical section.

The variable *turn* holds the identifier of the process whose turn it is and then the process waits to enter in the critical section. Next, a second transition (lines 16-18) is defined to specify the entering in the critical section.

The entrance in the critical section is granted for P_0 process if P_1 does not want to enter its critical section and if P_1 has given priority to P_0 by setting *turn* to 0. The third transition (lines 20-23) allows to do something during the critical section, updates the *flag* value and returns to the initial state.

Then, a main component is defined. It declares shared variables and instantiates two *Proc* process instances. Finally, the entry point for the verification is specified (line 38).

7.2 Expressing behavioral properties on FIACRE level

An intermediate language, introduced in the verification toolchain, will be considered for the DSML designer as the formal target level. So, it is mandatory that it offers the capability to express behavioral properties.

During the QUARTEFT project, a language was designed to specify behavioral and temporized properties at the FIACRE level. The designed language is implemented around two

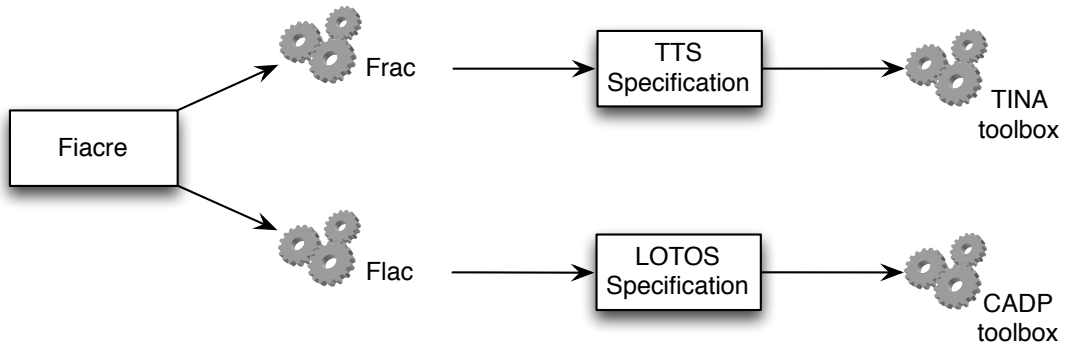

Figure 7.2 — The tooling around the FIACRE language

fundamental concepts: observable events and properties.

Observable events concern naming unambiguously components, processes, ports and variables of a FIACRE program and their related events which can be observed for each FIACRE element. To identify them, first, it is mandatory to identify the concerned process instance. It is done thanks to a *path* that identifies one single process instance. A FIACRE program should contain a main component which corresponds to the verification entry point. A process/component instance can be referred to the main component by an hierarchical manner using indices which allow to differentiate two instances of the same component/process [Abi12]. In Listing 7.1, "Main/1" means that we refer to the first instance in the *Main* component (line 32).

Then, the instance should be related to an observable event. Four kinds of events can be designated:

- **path/state s** returns true if the process instance identified by *path* is in state *s*.
- **path/value p** returns true if predicate *p* is true in the process instance identified by *path*.
- **path/tag t** is the set of transitions of the process instance identified by *path* bearing tag *t*. A tag is a kind of FIACRE statement inserted in a FIACRE process of form "#ident".
- **path/event p** is the set of transitions interacting on port *p* declared in the component identified by *path*.

Based on the observable events, a FIACRE program can be extended with properties. We can identify three kind of properties: general properties, LTL properties and real time properties. General properties can check if a system has a deadlock-free situation, if infinitely often an observable event is true or if an observable event is mortal. LTL properties use logic operators and temporal operators and real time properties that use the hierarchical classification borrowed from Dwyer [DAC98] extended with a notion of "timing modifiers" [Abi12]. A FIACRE description may include declarations of properties and "assert" directives.

In this thesis, we focus only on general and LTL properties. Listing 7.2 extends the FIACRE program shown in Listing 7.1 with general and LTL properties.

The first property named *ddlfree* (lines 4-5) verifies the absence of deadlock in the program.

The second property is a LTL property (named *mutex* in lines 8-9). It verifies whether the mutual exclusion occurs. It means that always both process instances (Main/1 and Main/2) cannot be in their critical sections (*CS* states) at the same time.

A third property is defined to verify the fairness of the program (lines 12-14). It verifies for each process instance if always when it sets its *flag* to true, then eventually this instance enters in critical section (*CS* state).

The fourth property (lines 17-18) verifies the *isIdle* concept: if the process P_0 does not set its *flag* to true, then it will never enter into the critical section.

Finally, we define a *infoften* property (lines 21-22) which verifies whether process P_0 infinitely often enters into the critical section.

```
1  /* Properties */
2
3  /* Absence of deadlock */
4  property ddlfree is deadlockfree
5  assert ddlfree
6
7  /* Mutual exclusion */
8  property mutex is ltl [] not ((Main/1/state CS) and (Main/2/state CS))
9  assert mutex
10
11 /* fairness */
12 property access is ltl ([] (Main/1/value flag[0] => <> Main/1/state CS)
13                       and [] (Main/1/value flag[1] => <> Main/2/state CS))
14 assert access
15
16 /* isIdle */
17 property isIdle is ltl (([] Main/1/value (not flag[0])) => ([] not Main/1/state CS))
18 assert idling
19
20 /* infoften */
21 property infoften is ltl ([] <> Main/1/state CS)
22 assert infoften
```

Listing 7.2 — Related behavioral properties on the implementation of Peterson's algorithm with FIACRE

Now, the complete FIACRE description is specified. FIACRE is also the source language of compilers into two verification toolboxes: TINA and CADP [GLMS11]. We rely in our experiments on the first toolbox. Using the FRAC compiler (the FIACRE compiler for the TINA toolbox), a FIACRE program is compiled into a generalization of TPN with data variables, guards, actions and priorities associated to transitions (Time Transition System (TTS)) that is one of the input formats accepted by the TINA toolbox (Figure 7.2).

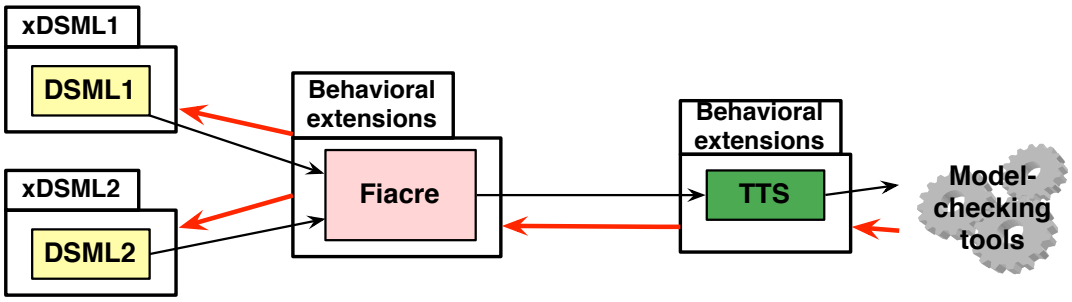

Figure 7.3 — The integration of FIACRE in the verification toolchain

7.3 Integrating the FIACRE language in the verification toolchain

The FIACRE language is shown as a high level formal language which allows to design a formal model and to specify behavioral properties. It is considered as a target formal language. To integrate it in the verification toolchain, several steps are required. The first one consists in providing different behavioral extensions conforming to the *Executable DSML pattern*. These extensions allow to capture different additional information captured during the execution (Figure 7.3).

Figure 7.4 shows an abstract view of the FIACRE EDMM. We have identified four kinds of events:

- the move of a process instance (*StateEvent*) which can leave a state (*ExitEvent*) or enter into another state (*EnterEvent*).
- the change of the value of a variable in a process instance (*VariableEvent*).
- a communication through a port (*PortEvent*). It can be shown as a *SynchronisationEvent*, a *ReceiveEvent* or a *SendEvent*.
- carrying a tag (*TagEvent*).

The FIACRE SDMM is shown in Figure 7.5. It includes two kinds of runtime information that can be generated during the execution of a FIACRE model: the current value (*Expression*) of a FIACRE variable (*Pattern*) and the current state (*currentState*) of an instance of a process. Referring to a FIACRE instance consists in collecting the ordered set of instances which leads from the main component to it (*Path*). Therefore, we add a reference (*currentState*) between the *Path* meta-class and the *StateDeclaration* meta-class.

7.4 Connecting the FIACRE level with the TINA toolbox

Integrating an intermediate language in a verification toolchain has significant advantages: (a) reducing the semantic gap between DSMLs and formal languages and (b) sharing parts

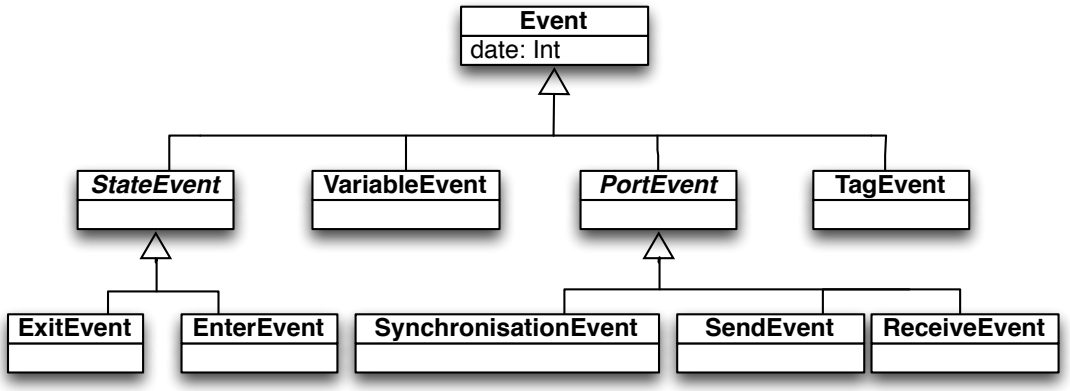

Figure 7.4 — Different kind of events in FIACRE

of the translation from DSMLs to formal languages (the intermediate to formal part). However, the feedback of the verification results should be made in two steps: a first one from the low-level formalism to the intermediate level and, then, from the intermediate level to the DSML.

In this section, we are interested in the feedback from low-level formal language (TTS) to the intermediate formal language (FIACRE) (the red arrow from TPN behavioral extensions to FIACRE behavioral extensions in Figure 7.3).

After performing the formal verification, it is mandatory to feedback this information, at first, to the FIACRE level then to the DSML one. However, this feedback is not trivial because it is not easy to find such a mapping between TTS specifications and FIACRE models.

Our proposal to feedback verification results, the FEVEREL language, could be an interesting candidate to perform this feedback but it is not the case as its current implementation supports only generating one DSML event from a formal event (1-to-1 mapping) or a partial format of the 1-to-n mapping which generates, from a formal event, a set of DSML events that are instances of the same DSML event meta-class. In addition, we do not know what is done during the compilation performed by the FRAC compiler. The amount of information handled during this compilation is important and FRAC is mostly a black-box tool. Furthermore, the semantic gap between TTS and FIACRE is wide and complicates the expression of such a mapping between both levels. Therefore, additional information are required: *traceability information*. These information contain what happened during the compilation. They give the correspondence between FIACRE elements (variables, ports, statements, etc.) and the generated ones in the TPN (states, transitions, etc.). In the following, we detail different required steps to feedback verification results from the TPN into the FIACRE level.

7.4.1 The generation of traceability information between FIACRE and TTS

The traceability information consists in storing a set of relations (named also mappings) between the corresponding source and target model elements in order to reuse them to verify

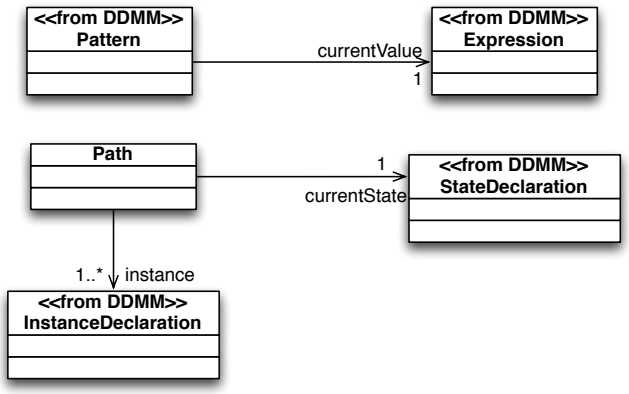

Figure 7.5 — The FIACRE SDMM

and validate software life-cycle. Several traceability approaches are proposed in the literature [GaG07]. For example, in [KPP06], authors introduce an approach named embedded traceability. In this one, the traceability elements are embedded inside the target models. For [Jou05], the traceability information are considered as a model, more precisely as an additional target model of a transformation program. For us, we choose the last one as it allows to separate different level of modeling.

To produce the traceability information, it is possible to extend the FRAC compiler to produce such an artifact which contains this mapping. As the FRAC compiler is not trivial to modify in order to generate an additional information to save a mapping, we considered it as a black box.

Furthermore, thanks to a specific option of the FRAC compiler (-G), it is possible to obtain an intermediate textual format of a FIACRE program that presents a hybrid TTS managed during the compilation. We name it *instantiated* FIACRE. It contains TPN specifications (transitions, states, priorities, ...) and data processing (guards, assignments, ...) and can thus be used to generate traceability links between TTS and the original FIACRE program.

Listing 7.3 shows the instantiated FIACRE program corresponding to the FIACRE program shown in Listing 7.1. First, it starts with the declaration of different data types (line 1). Then, it defines the main instantiated process which contains the whole traceability information. Its identifier corresponds to the different instantiated processes in the FIACRE program (line 3). Different instantiated states are declared using the **states** keyword (line 6). An instantiated state identifier follows this structure "p_i_**s**t" where *p* is an identifier that corresponds to a FIACRE process, *t* is also an identifier that corresponds to a state declared in the *p* process and *i* is an integer that corresponds to the rank of this process instance in the main component. Instantiated variables are defined using the **var** keyword (line 8) and initialized. They follow this structure "p_i_**v**t" where *p* and *i* correspond respectively to the FIACRE process and its instance, and *t* corresponds to the declared variable. Lines 10 and 11 show the initial states of process instances. Then, a set of instantiated transitions are defined. The signature of an instantiated transition (**Trans** keyword) has an identifier, a label, a root and a tag.

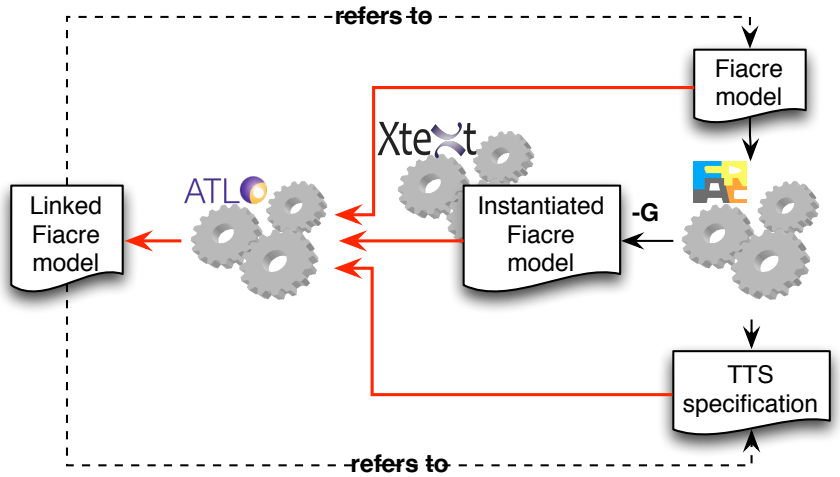

Figure 7.6 — The generation of the traceability information between FIACRE and TTS

```
type a0 is array 2 of bool

process Proc_2_Proc_1
    is

    states Proc_2_sCS, Proc_2_swaits, Proc_2_sidle, Proc_1_sCS, Proc_1_swaits, Proc_1_sidle

    var Main_1_vflag: a0 := [false,false], Main_1_vturn: char := 0

    init
    to Proc_1_sidle, Proc_2_sidle

    Trans::Proc_1_t0 & Main
      from Proc_1_sidle
       Main_1_vflag[0] := true;
       Main_1_vturn := 1;
       to Proc_1_swaits
      in [0,...[

    Trans::Proc_1_t1 & Main
      from Proc_1_swaits
       on not((Main_1_vflag[1] and (Main_1_vturn = 1)));
       to Proc_1_sCS
      in [0,...[

    Trans::Proc_1_t2 & Main
      from Proc_1_sCS
       Main_1_vflag[0] := false;
       to Proc_1_sidle
      in [0,...[

    Trans::Proc_2_t0 & Main
      from Proc_2_sidle
       Main_1_vflag[1] := true;
       Main_1_vturn := 0;
       to Proc_2_swaits
      in [0,...[

    Trans::Proc_2_t1 & Main
```

```
40        from Proc_2_swaits
41         on not((Main_1_vflag[0] and (Main_1_vturn = 0)));
42         to Proc_2_sCS
43        in [0,...[
44
45      Trans::Proc_2_t2 & Main
46        from Proc_2_sCS
47         Main_1_vflag[1] := false;
48         to Proc_2_sidle
49        in [0,...[
```

Listing 7.3 — The instantiated FIACRE model of the implementation of Peterson's algorithm with FIACRE

The identifier of the instantiated transition is then generated in the TTS description as a TPN transition. The label is an optional attribute. It corresponds to a FIACRE port if this transition has a port processing (a synchronisation via a port, the sending or reception of information). The root corresponds to the main component. In our case, it corresponds to the component *Main*. Finally, the tag identifies whether the transition carries a tag.

The instantiated transition body starts with the **from** keyword. It shows its initial states. The body contains a set of statements which correspond to performed actions (especially updates on variables' values) and eventually choose a next state using the jump statement **to**. The instantiated transition ends with a time interval for its execution. This interval has the same semantics as in TPN.

The generated textual artifact, instantiated FIACRE, shows interesting elements for the traceability. However, it is not sufficient to be considered as a traceability information. It is necessary to extend this artifact to refer to both sides, TPN and FIACRE. Figure 7.6 shows our approach to extend the instantiated FIACRE with the required information to obtain a complete traceability of the FRAC compiler. In fact, using Xtext, we define a textual grammar to parse the instantiated FIACRE. This one is derived from the FIACRE one.

Through an ATL transformation [JK06], based on the instantiated FIACRE model, the initial FIACRE model and the generated TPN model in the TTS specification, links are added between the FIACRE and TPN levels to generate a traceability model named *linked* FIACRE *model*. Its metamodel is similar to the instantiated one (generated by Xtext). It is enriched with references towards TPN and FIACRE appropriate elements.

Thanks to naming conventions and the hierarchical manner used to describe the composition of components and processes, traceability between FIACRE and TPN elements is made possible. A subset of the *Linked* FIACRE metamodel can be shown in Figure 7.7.

Now, the linked FIACRE model supports a traceability model in order to feedback verification results.

Let's explain what kind of information is added through this model transformation.

For each instantiated state (line 6 of Listing 7.3), we add two references: the first one refers to the i^{th} instance of the *p* process in the composition of the main component and the second one refers to the *t* state in the *p* process in the FIACRE specification. Each FIACRE state is prefixed with **s**. For example, for the instantiated state *Proc_1_sidle*, the first one refers to the first instance of the *Proc* process in the *Main* component (line 32 of Listing 7.1) and the second one refers to the *idle* state in the *Proc* process in the FIACRE specification (line 10 of

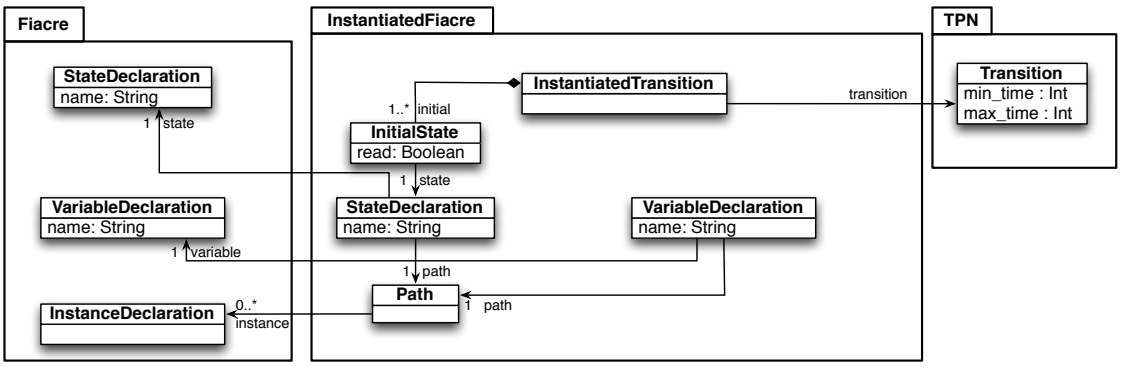

Figure 7.7 — A subset of the *Linked* FIACRE metamodel

Listing 7.1).

For instantiated variables, it is almost the same approach except that when *p* is a component, we refer to it. Each FIACRE variable is prefixed with **v**. For the *Main_1_vflag* instantiated variable, we add a reference to the *Main* component and another one to the *flag* shared variable in the same component (line 28 in Listing 7.1).

Finally, we handle on the instantiated transitions. For each one, we add a reference to the corresponding TPN transition.

In this way, we produce a complete traceability model between both sides. This traceability model allows to ease the feedback of verification results from the TPN level to the FIACRE level.

7.4.2 Feedback verification results on the FIACRE level

Once the traceability information is produced, it is easier to feedback verification results to the FIACRE level. We focus only on triggered events, instances of the FIACRE EDMM. Figure 7.8 illustrates our approach to produce a FIACRE scenario. In fact, the SELT model-checker produces a TPN counter-example for a violated property. Using the tooling shown in subsection 5.1, a TPN scenario is generated. Then, we define a M2M transformation using ATL which takes the traceability model (*linked* FIACRE *model*) and the TPN scenario, and generates a FIACRE scenario. This transformation takes each transition in the TPN scenario and collects the corresponding FIACRE events based on the traceability model. The generation of FIACRE events is guided by the type of the occurred statement in the traceability model. In fact, a **from** statement corresponds to leaving a FIACRE state (an instance of *ExitEvent* meta-class), an assignment statement corresponds to updating the value of a FIACRE variable (an instance of *VariableEvent* meta-class) and a **to** statement means the entering of an instance process into a new state (*EnterEvent*).

The instantiated FIACRE does not offer the required information to identify the port events (synchronisation, send and receive) on processes instances neither on subcomponents. It just gives the synchronisation on a port in the main component (an instance *SynchronisationEvent* meta-class). For the *TagEvent*, an instantiated transition can have a tag

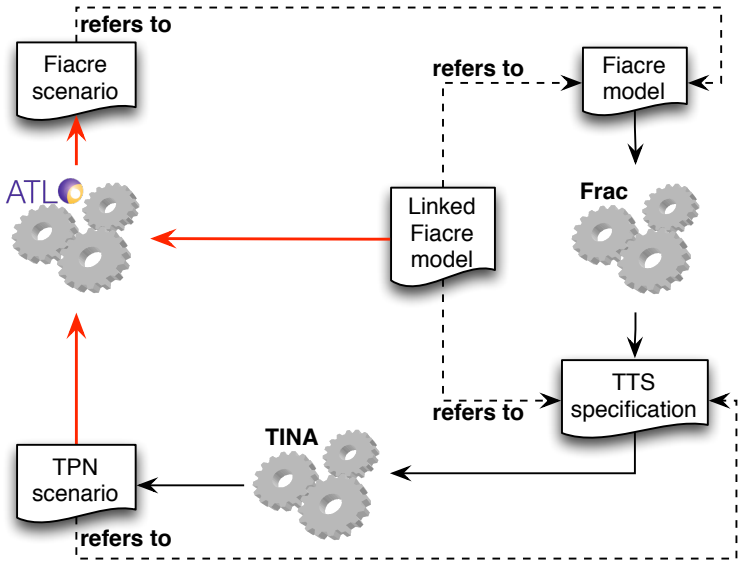

Figure 7.8 — The generation of the verification results on the FIACRE level

declaration which specifies carrying a tag, but, it does not specify the order of the execution. So, we decided to omit *TagEvents*.

```
operator ddlfree : prop
0.000s
TRUE
0.001s
operator mutex : prop
0.000s
TRUE
0.001s
operator access : prop
0.000s
TRUE
0.001s
operator idling : prop
0.000s
TRUE
0.001s
operator infoften : prop
0.000s
FALSE
  state 0: Proc_1_sidle Proc_2_sidle Main_w3
  -Proc_1_t0->
  state 1: Proc_1_swaits Proc_2_sidle {Main_1_vflag.at[0]} Main_1_vturn Main_w1
  -Proc_1_t1->
  state 2: Proc_1_sCS Proc_2_sidle {Main_1_vflag.at[0]} Main_1_vturn Main_w1
  -Proc_1_t2->
  state 3: Proc_1_sidle Proc_2_sidle Main_1_vturn Main_w3
  -Proc_2_t0->
  state 4: Proc_1_sidle Proc_2_swaits {Main_1_vflag.at[1]} Main_w3 Main_w2
  -Proc_2_t1->
  state 12: Proc_1_sidle Proc_2_sCS {Main_1_vflag.at[1]} Main_w3 Main_w2
  -Proc_2_t2->
* [accepting] state 13: Proc_1_sidle Proc_2_sidle Main_w3
  -Proc_2_t0->
```

```
34   state 14: Proc_1_sidle Proc_2_swaits {Main_1_vflag.at[1]} Main_w3 Main_w2
35   -Proc_2_t1->
36   state 16: Proc_1_sidle Proc_2_sCS {Main_1_vflag.at[1]} Main_w3 Main_w2
37   -Proc_2_t2->
38   state 13: Proc_1_sidle Proc_2_sidle Main_w3
39 0.001s
```

Listing 7.4 — Verification results generated in the TPN level for the Peterson's algorithm

Let's illustrate the feedback of verification results from the TPN level into the FIACRE level with the Peterson's algorithm. It corresponds to the FIACRE model shown in Listing 7.1 and the corresponding properties shown in Listing 7.2. The FRAC compiler generates a TTS specification from the full FIACRE program. Using the TINA toolbox and especially the SELT model-checker, the output of the verification is shown in Listing 7.4. This output shows that all properties are verified except the last one (*infoften* property). The counter-example (lines 20-38) shows an infinite loop (lines 32-38) which does not reach the appropriate element. It means that there is a possible execution where infinitely often the process P_0 does not enter into the critical section. Then, we perform different required steps to feedback verification results on FIACRE level. Listing 7.5 shows a subset of the corresponding FIACRE scenario. It is a succession of FIACRE events.

Another kind of verification result is the traces which is related to the SDMM part of the *Executable DSML pattern*. For the traces, the SELT output offers the possibility to obtain only FIACRE states (via TPN places). However, data values are lost. To obtain the complete traces, it is mandatory to simulate the obtained scenario via the command line stepper simulator *play*[1] of the TINA toolbox.

We are working on the integration of the output of the *play* simulator in order to obtain full traces on the FIACRE layer.

```
1 ExitEvent {path: Main/2, state: idle}
2 PatternEvent {pattern: flag[1], expression: true}
3 PatternEvent {pattern: turn, expression: 0}
4 EnterEvent {path: Main/2, state: waits}
5 ExitEvent {path: Main/2, state: waits}
6 EnterEvent {path: Main/2, state: idle}
7 ExitEvent {path: Main/2, state: idle}
8 PatternEvent {pattern: flag[1], expression: false}
9 EnterEvent {path: Main/2, state: idle}
```

Listing 7.5 — A subset of the FIACRE scenario corresponding to the verification results generated by SELT model-checker shown in Listing 7.4

7.5 Adapting the xSPEM toolchain to FIACRE

In this section, we apply the integration of an intermediate language on the xSPEM case study. Based on the proposed verification toolchain from chapter 3. As detailed in chapter 6 (section 6.3), this integration consists in changing the formal target domain. Therefore, it is necessary, at first, to update the *formalOperators* library corresponding to the FIACRE language. Then, a translational semantics should be implemented. It maps the xSPEM metamodel to the FIACRE level. It is mandatory to give the new implementation of the primitive

[1] http://projects.laas.fr/tina/manuals/play.html

queries (subsection 7.5.3) and the events mappings (subsection 7.5.4) for this new translational semantics as shown in chapter 6 (section 6.3).

In this section, we detail these different required elements to integrate the FIACRE language in the verification toolchain.

7.5.1 Connecting FIACRE properties capabilities with the TOCL tooling

FIACRE plays the role of the formal language from the DSML designer viewpoint. To reuse the TOCL editor, the definition of temporal operators traduction has to be updated. Listing 7.6 shows the implementation of FIACRE operators as strings showing their encodings.

```
helper def: always : String= '␣[]␣';
helper def: eventually : String= '␣<>␣';
helper def: next : String= '␣()␣';
helper def: "not" : String= '␣not␣';
helper def: "and" : String= '␣and␣';
helper def: "or" : String= '␣or␣';
helper def: "implies" : String= '␣=>␣';
helper def: until : String= '␣until␣';
helper def: release : String='␣release␣';
```

Listing 7.6 — The coding of FIACRE operators

7.5.2 Translational semantics XSPEM2FIACRE

The translational semantics consists in defining a mapping from the DSML, that is XSPEM, to the formal language, that is FIACRE.

Here is some rationale behind this translational semantics. We illustrate it with some elements in the FIACRE program corresponding to the updated version of the XSPEM model of Figure 2.2 with an additional *Computer* resource.

Each workdefinition is translated to one FIACRE process with the same name. Such a process is composed of three states (*notStarted*, *running* and *finished*) and two transitions (from *notStarted* to *running* and then from *running* to *finished*). Transition between the states depends on the worksequences and thus on the state of the predecessor workdefinition. Thus, it is necessary to store the current states of different workdefinitions.

Based on the QDMM of XSPEM, a FIACRE type called WDQueries was defined to represent the two queries on *WorkDefinition* of interest for the XSPEM end-user and to express causality constraints. It is a record type composed of the two boolean fields isStarted and isFinished.

```
type WDQueries is record          // from QDMM
        isStarted: bool,
        isFinished: bool
end
```

WDsQueries defines an array of WDQueries storing the state of all workdefinitions of an XSPEM process. It is an argument for every workdefinition process. This was defined

mainly to implement dependencies because a FIACRE process cannot inspect the current state of other processes.

```
type WDsQueries is array 4 of WDQueries end
```

Named constants are defined to ease the reading of the FIACRE program by avoiding the use of meaningless integers to identify a workdefinition.

```
const DesigningWD : int is 0
const ProgrammingWD : int is 1
const DocumentingWD : int is 2
const TestCaseWritingWD : int is 3
```

The WDsQueries variable is updated when a transition of a workdefinition process is fired. For example, on the transition from the *notStarted* state to the *running* state, the *isStarted* variable is set to *true*.

Furthermore, one workdefinition can only be started when required resources are available. As for workdefinitions, we have modeled resources queries as an array. Array elements are integers because there is no need of a record as there is only one query on *Resource* metaclass.

RessourceTab defines an array of integer storing the available count of each resource.

```
type RessourceTab is array 3 of int
```

As for workdefinitions, named constants are defined to ease identifying resources.

```
const DesignerR : int is 0
const DeveloperR : int is 1
const ComputerR : int is 2
```

xSPEM causality constraints are mapped into a FIACRE conditional statement that checks whether the FIACRE processes corresponding to the previous workdefinitions have reached the expected state. For example, because of the *startToStart* constraint between *Designing* and *Documenting*, conditional statement checks whether workdefinition *Designing* is started. It verifies also whether each required resource has the available amount to run this workdefinition. If the condition evaluates to false, nothing happens else the current state becomes *running*, the state of this workdefinition is updated, and the available resources are decreased. The following process gives the *Programming* workdefinition translated into FIACRE specification.

```
process Programming
        (&WorkDefinition: ProcessWDQueries, &Ressource: RessourceTab)is
states notStarted, running, finished

from notStarted
        if ( WorkDefinition[$(DesigningWD)].isFinished and
             WorkDefinition[$(TestCaseWritingWD)].isStarted and
             Ressource[$(DeveloperR)]>=2 and
             Ressource[$(ComputerR)]>=2 )
        then
         Ressource[$(DeveloperR)] := Ressource[$(DeveloperR)] - 2;
         Ressource[$(ComputerR)] := Ressource[$(ComputerR)] -2;
         WorkDefinition[$(ProgrammingWD)].isStarted:= true;
         to running
        else
         loop
```

```
        end if

from running
        WorkDefinition[$(ProgrammingWD)].isFinished:= true;
        Ressource[$(DeveloperR)] := Ressource[$(DeveloperR)] + 2;
        Ressource[$(ComputerR)] := Ressource[$(ComputerR)] +2;
        to finished
```

The FIACRE component *Main* consists in instantiating one FIACRE process for each workdefinition in the XSPEM process (here four processes for *Designing*, *Programming*, *Documenting* and *TestCaseWriting*) with the array that stores workdefinitions' states (initially all workdefinitions are not started and not finished). In addition, it initializes available amounts for different resources.

```
component Main is
var

 WorkDefinition: ProcessWDQueries := [{isStarted=false,isFinished=false},
                                      {isStarted=false,isFinished=false},
                                      {isStarted=false,isFinished=false},
                                      {isStarted=false,isFinished=false}],
 Ressource : RessourceTab := [2,3,4]

par
   Designing (&WorkDefinition,&Ressource)
|| Programming (&WorkDefinition,&Ressource)
|| Documenting (&WorkDefinition,&Ressource)
|| TestCaseWriting (&WorkDefinition,&Ressource)
end
```

This translational semantics is defined as a model to model (M2M) transformation expressed in ATL [JK06]. Then, using the textual grammar of FIACRE defined using Xtext, we generate the FIACRE textual model, the input of the FRAC compiler.

7.5.3 Defining and translating TOCL properties

Once the translational semantics is defined, it is mandatory to define different primitive queries of the verification toolchain.

According to our approach (chapter 6), when the target formal property language changes, primitive queries should be updated. Non-primitive queries and behavioral properties do not change because they only depend on the DSML and other queries. Listing 7.7 shows the implementation of XSPEM queries. These queries ask whether a corresponding workdefinition is in the appropriate state based on the defined translational semantics.

```
1 module spem;
2 import  "http://Spem" as SPEM
3
4 // SPEM queries
5 context SPEM!WorkDefinition def : isFinished(): String=
6           'Main/1/value_WorkDefinition[$(' + self.name + 'id)].isFinished';
7
8 context SPEM!WorkDefinition def : isStarted(): String=
9           'Main/1/value_WorkDefinition[$(' + self.name + 'id)].isStarted';
```

Listing 7.7 — Formalization of SPEM queries based on the translational semantics defined on FIACRE

Based on the defined tooling `TOCL2ATL`, a model-to-text transformation is generated. This later takes a xSPEM model and generates properties at the FIACRE level. Based on the SPEM model defined in Figure 2.2, the corresponding generated properties are shown here:

```
property willNeverFinish is ltl
([](not(Main/1/value WorkDefinition[$(DesigningWD)].isFinished
        and Main/1/value WorkDefinition[$(ProgrammingWD)].isFinished
        and Main/1/value WorkDefinition[$(DocumentingWD)].isFinished
        and Main/1/value WorkDefinition[$(TestCaseWritingWD)].isFinished
)))

property willEventuallyFinish is ltl
<> (Main/1/value WorkDefinition[$(DesigningWD)].isFinished
    and Main/1/value WorkDefinition[$(ProgrammingWD)].isFinished
    and Main/1/value WorkDefinition[$(DocumentingWD)].isFinished
    and Main/1/value WorkDefinition[$(TestCaseWritingWD)].isFinished)
```

Once the complete FIACRE specification is generated, different translation and formal verification steps are performed.

7.5.4 The feedback of verification results

The SELT model-checker shows that the first property, `willNeverFinish`, does not hold and a TPN counter-example is generated (Listing 7.8). Conforming to the process shown in subsection 7.3, the verification results are produced at the FIACRE level.

```
1 FireTransitionEvent Designing_1_t0
2 FireTransitionEvent Designing_1_t2
3 FireTransitionEvent Documenting_1_t0
4 FireTransitionEvent Documenting_1_t2
5 FireTransitionEvent TestCaseWriting_1_t0
6 FireTransitionEvent Programming_1_t0
7 FireTransitionEvent Programming_1_t2
8 FireTransitionEvent TestCaseWriting_1_t2
```

Listing 7.8 — A TPN scenario generated by SELT model-checker

Listing 7.9 shows a concrete view of the corresponding FIACRE scenario. It represents the verification results shown for the DSML designer. It contains a set of FIACRE events, instances of FIACRE EDMM meta-classes (Figure 7.4). Each five events (lines 1-5, lines 6-10, etc.) corresponds to the i^{th} TPN event.

Therefore, the DSML designer can use them to generate DSML verification results.

```
1  ExitEvent {path: Main/1, state: notStarted}
2  PatternEvent {pattern: Ressource[0], expression: Ressource[0] − 2}
3  PatternEvent {pattern: Ressource[2], expression: Ressource[2] − 2}
4  PatternEvent {pattern: WorkDefinition[0].isStarted, expression: true}
5  EnterEvent {path: Main/1, state: running}
6  ExitEvent {path: Main/1, state: running}
7  PatternEvent {pattern: WorkDefinition[0].isFinished, expression: true}
8  PatternEvent {pattern: Ressource[0], expression: Ressource[0] + 2}
9  PatternEvent {pattern: Ressource[2], expression: Ressource[2] + 2}
10 EnterEvent {path: Main/1, state: finished}
11 ExitEvent {path: Main/3, state: notStarted}
12 PatternEvent {pattern: Ressource[0], expression: Ressource[0] − 1}
13 PatternEvent {pattern: Ressource[2], expression: Ressource[2] − 1}
```

```
PatternEvent {pattern: WorkDefinition[2].isStarted, expression: true}
EnterEvent {path: Main/3, state: running}
ExitEvent {path: Main/3, state: running}
PatternEvent {pattern: WorkDefinition[2].isFinished, expression: true}
PatternEvent {pattern: Ressource[0], expression: Ressource[0] + 1}
PatternEvent {pattern: Ressource[2], expression: Ressource[2] + 1}
EnterEvent {path: Main/3, state: finished}
ExitEvent {path: Main/4, state: notStarted}
PatternEvent {pattern: Ressource[1], expression: Ressource[1] - 1}
PatternEvent {pattern: Ressource[2], expression: Ressource[2] - 2}
PatternEvent {pattern: WorkDefinition[3].isStarted, expression: true}
EnterEvent {path: Main/4, state: running}
ExitEvent {path: Main/2, state: notStarted}
PatternEvent {pattern: Ressource[1], expression: Ressource[1] - 2}
PatternEvent {pattern: Ressource[2], expression: Ressource[2] - 2}
PatternEvent {pattern: WorkDefinition[1].isStarted, expression: true}
EnterEvent {path: Main/2, state: running}
ExitEvent {path: Main/2, state: running}
PatternEvent {pattern: WorkDefinition[1].isFinished, expression: true}
PatternEvent {pattern: Ressource[1], expression: Ressource[1] + 2}
PatternEvent {pattern: Ressource[2], expression: Ressource[2] + 2}
EnterEvent {path: Main/2, state: finished}
ExitEvent {path: Main/4, state: running}
PatternEvent {pattern: WorkDefinition[3].isFinished, expression: true}
PatternEvent {pattern: Ressource[1], expression: Ressource[1] + 1}
PatternEvent {pattern: Ressource[2], expression: Ressource[2] + 2}
EnterEvent {path: Main/4, state: finished}
```

Listing 7.9 — A FIACRE scenario corresponding to the verification results generated by SELT model-checker shown in Listing 7.8

Using the FEVEREL language, he must define a mapping between FIACRE events and the corresponding ones in the XSPEM level. Listing 7.10 shows a possible implementation of this mapping.

```
import "http://spemSemantics/1.0" as DSMLSemantics
import "http://fiacreSemantics/1.0" as FormalSemantics
import "http://spemDDMM/1.0" as DSMLAS
import "http://www.topcased.org/fiacre/xtext/Fiacre" as FormalAS

events mapping swd2t:
        DSMLEvent swd: DSMLSemantics.StartWD(
                date <- ev1.date
        )
        maps
        FormalEvent ev1 :FormalSemantics.EnterEvent (
                ev1.state.name = 'running' and
                FormalAS!Model.allInstances()->first().root.body.blocks
                ->indexOf(ev1.path.instances->first())
                =
                DSML!Process.allInstances()->first().workDefinitions
                ->indexOf(swd.workdefinition)
        )
end events mapping

events mapping  fwd2te:
        DSMLEvent fwd: DSMLSemantics.FinishWD (
                date <- ev2.date
        )
        maps
        FormalEvent ev2: FormalSemantics.EnterEvent (
                ev2.state.name = 'finished' and
                FormalAS!Model.allInstances()->first().root.body.blocks
```

```
29                  ->indexOf(ev2.path.instances->first())
30                  =
31                  DSML!Process.allInstances()->first().workDefinitions
32                  ->indexOf(fwd.workdefinition)
33          )
34 end events mapping
```

Listing 7.10 — The definition of events mappings using FEVEREL in the case-study of the verification of SPEM models using FIACRE

We choose to refer to FIACRE *EnterEvent* instances to generate the corresponding ones in the XSPEM level. To generate a *StartWD* (lines 7-8) (respectively *FinishWD* (lines 22-23)) event in the XSPEM side, we choose to refer to *EnterEvent* whose state is *running* (line 12) (respectively *finished* (line 27)) and the index of the unique instance of *path* in the composition of *Main* component corresponds to the index of the related workdefinition in the XSPEM process.

This FEVEREL model is then transformed using the FEVEREL tooling into an ATL model transformation which takes the FIACRE scenario as input and generates a XSPEM scenario shown in Listing 7.11.

```
1 StartWD Designing
2 FinishWD Designing
3 StartWD Documenting
4 FinishWD Documenting
5 StartWD TestCaseWriting
6 StartWD Programming
7 FinishWD Programming
8 FinishWD TestCaseWriting
```

Listing 7.11 — A SPEM scenario generated from the FIACRE one shown in Listing 7.9

8 Formal verification of PLC programs

Résumé

Le but de ce chapitre est de valider notre approche pour tirer parti de l'activité de vérification pour tout utilisateur final d'un DSML. On vise alors un nouveau domaine qui est les automates programmables industriels (APIs), ou les Programmable Logic Controllers en anglais (PLCs). un API est un type particulier d'ordinateur utilisé pour automatiser les processus industriels.

Les APIs sont des ordinateurs industriels dédiés, conçus pour contrôler des machines et des processus. L'aspect critique des éléments conçus rend la plupart des défaillances catastrophiques pour la sécurité des équipements et des humains. De ce fait, il est nécessaire de les vérifier afin de détecter les éventuels problèmes le plus tôt possible durant le processus de développement. Actuellement, l'activité de vérification des APIs s'effectue en utilisant la technique de test qui est extrêmement coûteuse en temps d'une part et très incomplète pour les systèmes complexes d'autre part. Par conséquent, les techniques de vérification formelle peuvent être considérées comme un candidat valable pour assurer la sûreté et l'efficacité des systèmes contrôlés.

Le Langage Ladder, ou Ladder Diagram (LD) en anglais, est le langage de modélisation le plus utilisé pour concevoir les APIs. Il possède une structure graphique qui rend la détection des erreurs plus difficile. Par conséquent, l'utilisation des méthodes formelles (par exemple la vérification formelle par exploration exhaustive des modèles) devient obligatoire pour la sûreté et l'exactitude des systèmes conçus.

Dans des travaux précédents, une chaîne de transformation a été définie à partir de LD vers le langage intermédiaire FIACRE afin de vérifier des propriétés comportementales génériques liées au langage LD. Par contre, ces propriétés sont exprimées directement en LTL et les résultats de vérification sont uniquement générés au niveau formel. Cette intégration partielle des méthodes formelles ne correspond pas aux attentes des utilisateurs de LD parce que la notation formelle des résultats de vérification est loin des pratiques industrielles. En outre, pour le concepteur LD, exprimer les propriétés comportementales au niveau formel ne correspond pas à ses propres capacités.

Dans ce chapitre, on applique notre approche à la vérification des APIs. Cela consiste à formaliser des propriétés génériques au niveau LD en utilisant l'outillage TOCL, générer

automatiquement les propriétés formelles correspondantes et remonter les résultats de vérification pour qu'ils soient utilisables par les utilisateurs finaux.

Ce chapitre introduit d'abord les notions d'API et de LD illustrées avec une étude de cas industrielle ainsi que les propriétés comportementales attendues. Elles seront modélisées au niveau LD. On expliquera brièvement la chaîne de vérification ad-hoc existante pour le langage LD. On expliquera alors l'intégration de nos contributions pour étendre la chaîne outillée de vérification et pour cacher totalement les aspects formels pour les utilisateurs finaux du langage LD. Cette intégration s'appuie sur l'application du patron de métamodélisation sur le métamodèle du langage LD afin d'identifier les requêtes, les événements et les états qui peuvent être observés à ce domaine.

The purpose of this chapter is to validate our approach for leveraging verification activities to any DSML end-user. We thus address a new domain, the Programmable Logic Controllers (PLCs). A PLC is a special purpose computer used to automate industrial processes.

PLCs are industry-dedicated computers designed to control machines and processes. The critical aspect of the designed elements makes most failure occurrence catastrophic for equipments and human safety. Thereby, it is mandatory to verify them in order to detect eventual issues as early as possible during the development process. Currently, the verification activity of PLCs is performed using testing which is an extremely costly and time-consuming method on the one hand and highly incomplete for complex systems on the other hand. Therefore, formal verification techniques can be considered as an meaningful candidate to ensure safety and efficiency to the controlled systems.

Ladder Diagram (LD) is one of the most used modeling languages to design PLCs. It has a graphical structure which makes error detection more difficult. Hence, the use of formal methods (model-checking for example) becomes mandatory for to the safety and correctness of the designed systems.

In previous works [FdQdS+11, FDQDR+11], a transformation chain has been defined from LD to the FIACRE intermediate language in order to verify generic behavioral properties related to the LD domain. However, these properties are expressed directly using LTL and the verification results are only generated in the formal level. This current partial integration of formal methods does not correspond to the LD users expectations because the formal notation of the verification results is far from the industrial practices. In addition, for the LD designer, expressing behavioral properties in the formal side does not correspond to his own capabilities.

In this chapter, we apply our contributions to extend the integration of formal methods for the verification of PLC programs. It consists in formalizing generic properties at the LD level using the TOCL tooling, generating automatically their corresponding formal properties and managing verification results in order to be understood by LD system designers.

We first introduce the PLC and the LD notions illustrated with an industrial case-study, as well as the expected behavioral properties to be modelled for LD. Then, we briefly show the verification activity introduced for the LD language proposed in [FdQdS+11, FDQDR+11]. Finally, we show how we can apply our contributions to extend the verification toolchain and hide the whole formal aspects for LD end-users. This integration is obviously based on the application of the *Executable DSML pattern* on the LD metamodel to identify the queries, events and states that can be observed on this side.

8.1 Specification of PLC programs

In this section, we introduce the PLCs, the IEC 61131-3 standard [Com03] and the LD programming language. We illustrate it with a Control System use-case shown in [Ben08]. We discuss the behavioral properties which must be verified for the LD domain.

8.1.1 PLCs and the IEC 61131-3 standard

A PLC is a special purpose industrial computer used for automation of industrial processes. A PLC program has several inputs and outputs. Designing a PLC program helps to control the state of the outputs depending on the configuration of the related inputs and its internal state. It is designed to support severe conditions as electrical noise, and resistance to vibration and impact [BCC+08].

The PLC execution follows a cycle started by copying the state of the whole inputs into the memory. Then, the core program runs and produces in the memory a temporary table of all outputs. When this program finishes, the table is written to the outputs and a new cycle starts. This cycle repeats as long as the PLC is running [BCC+08].

The IEC 61131-3 is an international standard of the International Eletrotechnical Commission that regulates the programming languages for PLCs. It introduces five different programming languages: Instruction List (IL), Structured Text (ST), Function Blocks Diagrams (FBD), Sequential Function Chart (SFC) and Ladder Diagram (LD). A PLC program can be written using one or more of these languages. Theirs semantics are not rigorously defined, and certain definitions can contain several ambiguities.

8.1.2 Ladder Diagram (LD)

The LD language is the most used language for programming PLCs. It is one of the two graphical languages described by the IEC 61131-3 standard, and it is based on the relay logic.

Figure 8.1 shows a simple example of a LD program. A LD program has two vertical rails and a set of horizontal lines (*rungs*) between them. Each rung is read from the left, containing input instructions, to the right which represents the output instruction. Each rung represents a boolean equation. Two kinds of constructs can be identified: contacts, named also relays, and coils. The relays, represented graphically by two bars `| |`, are input instructions associated to a program variables. They participate in forming the boolean function to calculate the new value of their rung outputs, using *and* boolean operation when they are placed in series and *or* boolean operation when they placed in parallel. A diagonal line is placed in the middle of symbols as in `|/|` to indicate that the negated value of the variable is used. The coils, shown as two parentheses `( )`, represent the output variables, and they, unlike the relays, do modify the value of the associated variables [Car12]. In addition to the shown simple elements of LD, several complex ones like function blocks can be found in the complete documentation [Com03].

The LD program shown in Figure 8.1 represents the boolean equations:
$C = (A \vee C) \wedge \neg B$ and $D = C$.

To model LD programs, a LD metamodel shown in Figure 8.2, is proposed to define the concepts of a *Program* composed of (1) a set of variables (*Variable*) and (2) a set of rungs (*Rung*). The *Variable* meta-class represents a declared LD variable in the LD program. Only boolean variables are handled. It is characterized by a *name*, a *value* and a kind (*inOutKind*) to specify whether it is an input, an output or a memory (*mem*).

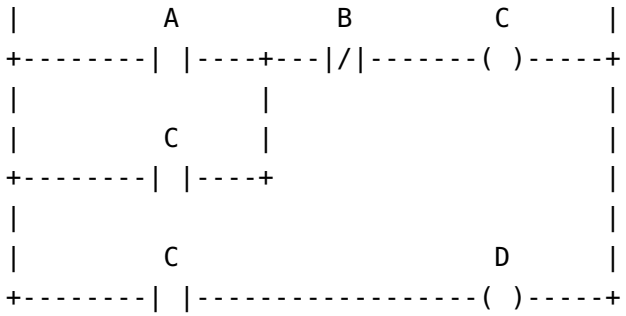

Figure 8.1 — Simple LD example

A LD rung (*Rung*) is composed of one or more elements (*Element*) and semi-rungs (*SemiRung*). Two different rungs cannot share the same elements. A rung has only an integer attribute (*index*). it allows to order the rungs in the LD program. A *SemiRung* allows to model each part of a LD rung. They are introduced to deal with LD programs that use Functional Blocks. Each semi rung has paths (*Path*).

Two kinds of elements are defined: *BasicElement* and *ComplexElement*. A *BasicElement* can be a *Coil* which is always a result (*Result*) or a *Contact* which is always an operator (*Operator*). Each basic element is characterized by a *kind* (*CoilKind* for a *Coil* and *ContactKind* for a *Contact*). A *ComplexElement* can be a function or a function block in a LD program. It is characterized by a *kind* to specify the type of the function. The *Functions* enumeration can be extended to include additional standard functions and function blocks. Complex elements are composed of inputs (*Input*), outputs (*Output*) and internal variables (*InternalVariable*). Like any PLC program, the execution of the LD diagram is shown as consecutive cycles, called execution cycle or just *Scan*, composed of three steps: The first one consists in reading the input variables from the sensors connected with the LD program. Then, a calculating process is performed. Related to the rung input function and the values of input variables stored in the previous step, the new values of intern memory and output variables of the program are generated. This computation is made following the rungs from top to bottom, and from left to right. Finally, the third step is the writing of output variables - that have been calculated before - in PLC actuators.

A designed LD program may not be the appropriate one which reflects the expected specification of the system. So, a LD program may contain several errors that affect its execution. Therefore, it is necessary to introduce verification activities as early as possible to ensure the correctness of the designed program.

Two types of properties are identified: generic properties and specific properties. Generic properties are only based on the LD concepts. One of the important generic properties to be verified on an LD program is *the absence of race conditions* [AFS98]. A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations in parallel, because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly. A race condition occurs in an LD program when under fixed inputs and function block states, one or more outputs keep changing their values.

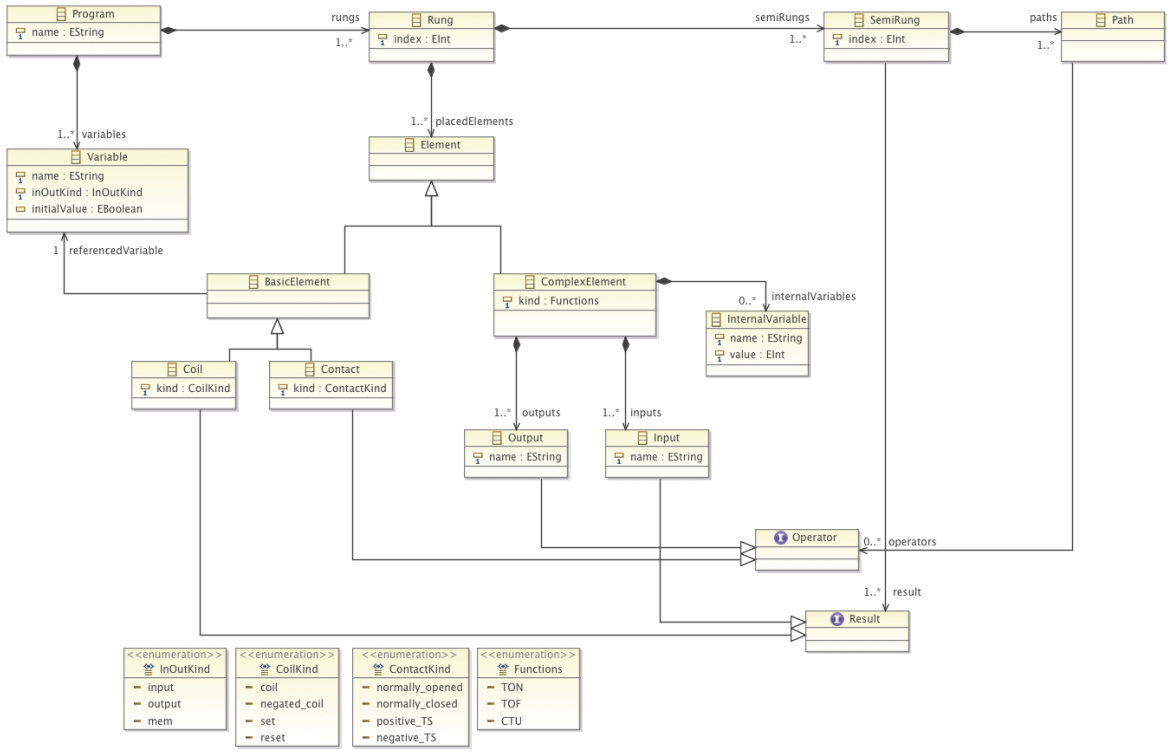

Figure 8.2 — LD metamodel

To verify this kind of property a possible formalization is proposed in [Ben08, FDQDR+11]. First, a *stability* concept for inputs and outputs is proposed. A LD variable is stable means that its value does not change: it is always *True* or always *False*. Using LTL, a possible definition is the following:

Definition 7. *A LD variable called v is stable if ((□ vTrue ∨ □ vFalse))*

A LD program is free of race conditions if, when the inout variables are kept stable, all output variables and memories will stabilize.

Definition 8. *A LD program is free of race condition if* $\square\ (stable_{inputs} \implies \Diamond\ stable_{outputs})$. $stable_{inputs}$ *represents a logical conjunction between all input variables stability condition and* $stable_{outputs}$ *represents a logical conjunction between the stability condition of all output and memories.*

In the LD example of Figure 8.3, the variable *A* is an input, *C* and *D* are memories, *B* and *E* are outputs. It can be seen that even if *A* is kept stable, the variables *C*, *D* and *E* will not stabilize, thus it is an example of race condition. This kind of problem is sometimes difficult to detect with traditional techniques, and bugs not detected during the test period can be very costly to correct later.

Specific properties are related to the specific system being designed. In [Car12], a possible specification of these properties has been proposed.

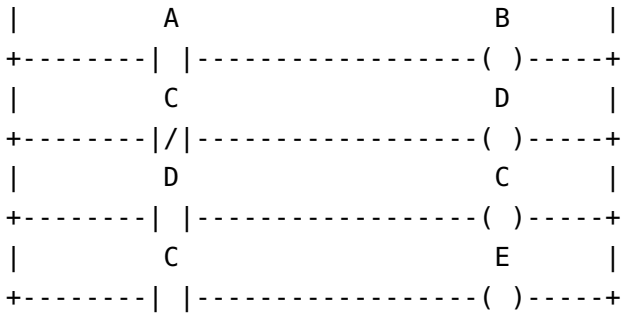

Figure 8.3 — Simple example of races in LD

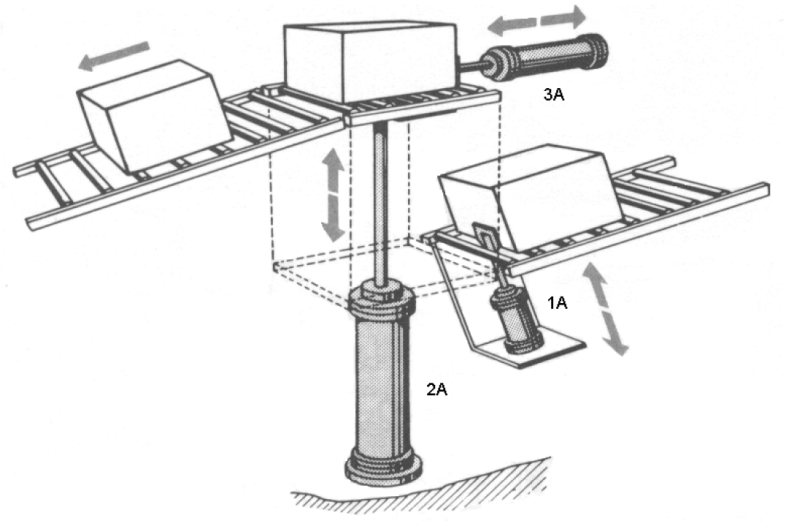

Figure 8.4 — Draw of the elevation system

8.1.3 A Control System Example

To illustrate the concepts of modeling with LD language, we describe a control system used in [Ben08]. It represents an elevation system with pneumatic actuators inserted in a box transportation plant as showed in Figure 8.4. When a box arrives in the lower conveyor (sensor 1S0), the cylinder 1A retracts, allowing the box to slide over the elevation table (sensor 2S0). After that, the cylinders 2A, 1A and 3A will sequentially extend, occasioning the elevation and expulsion of the box in the upper conveyor. Finally the cylinders 2A and 3A retract simultaneously. In this state the system is ready to receive an other box.

In addition to sensors 1S0 and 2S0, the elevation system is composed by the instrumentation presented in Figure 8.5. The activation of coils Y1, Y3 and Y5 results in the extension of cylinders 1A, 2A and 3A respectively, while the activation of coils Y2, Y4 and Y6 results in the retraction of these cylinders. Sensors 1S2, 2S2 and 3S2, are activated when cylinders 1A, 2A and 3A respectively are completely extended. Sensors 1S1 and 2S1 are activated when cylinders 1A and 2A are completely retracted.

In Figure 8.6 is presented an LD program that can be used to control this plant.

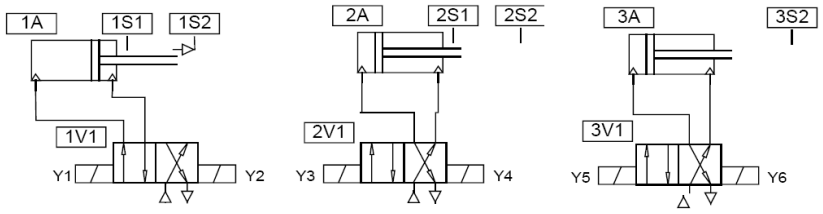

Figure 8.5 — Sensors and actuators of the elevation system

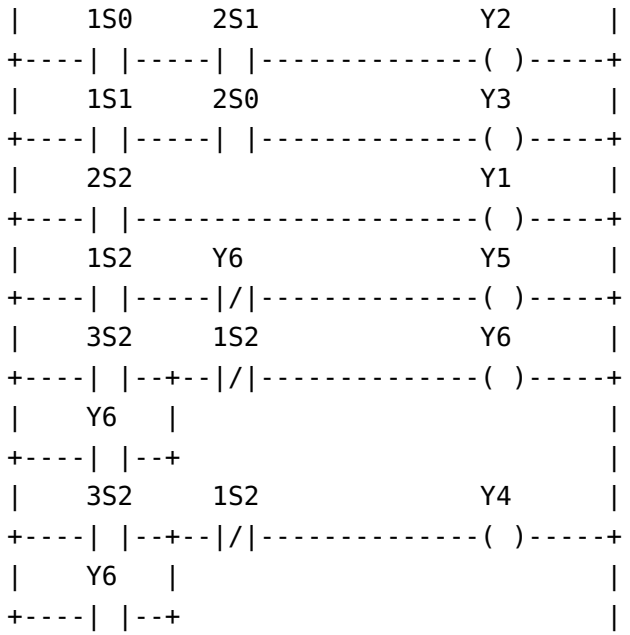

Figure 8.6 — LD program to control the elevation system

8.2 Modeling and Verification of PLC programs

We now describe the proposed approach in [FDQDR+11] for the formal verification of PLC programs. First, we explain the architecture of a system with FIACRE. Then, we present the proposed PLC verification toolchain. Finally, we illustrate the proposed approach with the control system example shown in the subsection 8.1.3.

8.2.1 Modeling PLC programs with the FIACRE language

Figure 8.7 shows the architecture of a system defined with FIACRE. It is the composition of the PLC controller with the *Plant*. They are designed as FIACRE components. FIACRE is based on synchronous component communication. So, it is mandatory to introduce an intermediate *Glue* component to ensure the asynchronous behavior between the plant and the controller.

For the PLC execution cycle, three activities are identified: input reading, program execution and output writing. They occur sequentially. First, the PLC reads sensor device information available on its input interface and in the end, writes actuator device commands on

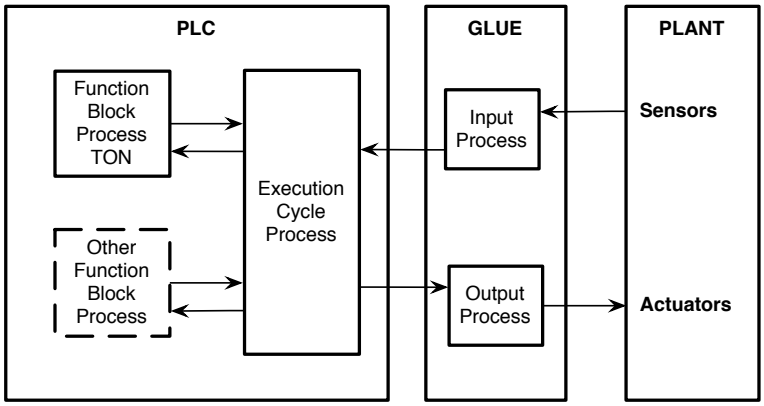

Figure 8.7 — Modeling the system in FIACRE

its output interface, linked with the plant to be controlled.

The main component in FIACRE shows the complete system. It is a parallel composition of *PLC*, *plant* and *glue* components. The PLC component is obtained by instantiating the execution cycle process which includes basic LD elements (as rungs, contacts, coils) and function blocks (FB) processes (as timers, counters, etc.). The FIACRE code of the PLC component is obtained from the translation of IEC language representations. The glue component is made up of an input process and an output process representing respectively sensor and actuator interfaces. The plant component is the result of the composition of processes which represent the behavior of each equipment.

8.2.2 Existing PLC Verification toolchain

The existing PLC verification toolchain is the following one. Using a PLC editor, PLC programs can be defined. Then, using ATL, a translator is defined which maps the LD model into a FIACRE specification. It is linked by a composer module with the plant and glue FIACRE models, resulting in the FIACRE system representation. Finally, using the FRAC compiler, the full FIACRE specification is compiled into a TTS specification [FDQDR+11]. As future works, it can be interesting to define a DSML to model the plant and the glue and then translate them to FIACRE.

In [FDQDR+11], behavioral properties are specified with LTL despite it is not the appropriate manner to specify them for the PLC program designer because temporal logic is far from industrial practices. So, the aim is to be able to automatically generate these formulas from the desired property specifications, written by process and control engineers in accordance with the practice of each application domain.

In our work, we focus only on the generic part which concerns the translational approach of LD models into FIACRE specifications. First, the LD designers choose to define an *indexOut* type to identify uniquely each output variable. An *arrayIn* type is defined to store the current value for each input variable and an *arrayOut* type is defined to store the current value for each output variable.

```
type indexOut is  0 .. 4
type arrayIn is array 7 of bool
type arrayOut is array 5 of bool
```

Then, *Input* and *Output* processes are defined.

```
process Input [ sendVar : out bool ]  is states varTrue, varFalse
init to varFalse

from varFalse
        select
                sendVar ! false ;loop
                []
                sendVar ! true ; to varTrue
        end

from varTrue
        select
                sendVar ! true ; loop
                []
                sendVar ! false ; to varFalse
        end

process Output [ receiveVar : in arrayOut ] ( arrayIndexVar :  indexOut )
is states varTrue , varFalse
var outputsVarsArray : arrayOut

init to varFalse

from varFalse
        receiveVar ? outputsVarsArray ;
        if outputsVarsArray [ arrayIndexVar ] then
                to varTrue
        else
                loop
        end

from varTrue
        receiveVar ? outputsVarsArray ;
        if outputsVarsArray [ arrayIndexVar ] then
                loop
        else
                to varFalse
        end
```

Each process has two states: *varTrue* for the true value and *varFalse* for the false value.

Then, the input glue process *InputGlue* is defined. It receives the sensor values from the plant component, stores them and sends them to the PLC component (more precisely to the execution cycle process) at the beginning of each cycle.

The output process receives output values obtained from PLC components at the end of the execution cycle process, stores them and sends them to the actuators of the plant components. The designers choose to implement it when the specific part of the plant is added.

In addition, the execution cycle process *Scan* is defined. The PLC execution cycle process begins with a reading transition which receives an array of input data from the glue input process. The rungs of the LD are represented as states. Transitions allow moving from one rung to another; a zero delay is assumed for each transition. From the last rung of the LD, an array of output data is sent to the glue output process in a writing transition which is followed by a silent transition (wait) corresponding to the restart of cycle, after a delay corresponding to its period.

```
process Scan [ portInputs : in arrayIn , portOutputs : out arrayOut ]
```

```
is states initial , writing , final , rung_1 , rung_2 , rung_3 ,rung_4 , rung_5 , rung_6
var
varsIn : arrayIn, v1S0 : bool := false, v2S1 :bool := false, v1S1 : bool := false,
v2S0 : bool := false, v2S2 : bool := false, v1S2 : bool := false, v3S2 : bool := false,
Y1 : bool := false, Y2 :bool := false, Y3 : bool := false, Y4 : bool := false,
Y5 : bool := false, Y6 : bool := false

init to initial

from initial
        portInputs ? varsIn ;
        v1S0 := varsIn [ 0 ] ;
        v2S1 := varsIn [ 1 ] ;
        v1S1 := varsIn [ 2 ] ;
        v2S0 := varsIn [ 3 ] ;
        v2S2 := varsIn [ 4 ] ;
        v1S2 := varsIn [ 5 ] ;
        v3S2 := varsIn [ 6 ] ;
to rung_1

from writing
        portOutputs ! [ Y1 , Y2 , Y3 , Y4 , Y5 ] ;
to final

from final
        wait [ 1 , 1 ] ;
        to initial

from rung_1
        wait [ 0 , 0 ] ;
        Y2 := v2S1 and v1S0 ;
to rung_2

from rung_2
        wait [ 0 , 0 ] ;
        Y3 := v2S0 and v1S1 ;
to rung_3

from rung_3
        wait [ 0 , 0 ] ;
        Y1 := v2S2 ;
to rung_4

from rung_4
        wait [ 0 , 0 ];
        Y5 := not Y6 and v1S2 ;
        to rung_5

from rung_5
        wait [ 0 , 0 ] ;
        Y6 := Y6 and not v1S2 or not v1S2 and v3S2 ;
        to rung_6

from rung_6
        wait [ 0 , 0 ] ;
        Y4 := Y6 and not v1S2 or not v1S2 and v3S2 ;
        to writing
```

The *Scan* process is instantiated in the *PLC* component.

```
component PLC [ portInputs : in arrayIn , portOutputs : out arrayOut ] is
par * in
        -> Scan [portInputs , portOutputs ]
end
```

In addition, an *Inputs* component is defined. It instantiates the *InputGlue* process with a set of *Input* process. Each instance corresponds to an input LD variable.

```
component Inputs [ writeInputs : out arrayIn ,readOutputs : in arrayOut ] is
port
v1S0Port : in out bool in [ 0 , 0 ] ,
v2S1Port : in out bool in [ 0 , 0 ] , v1S1Port : in out bool in [ 0 , 0 ] ,
v2S0Port : in out bool in [ 0 , 0 ] , v2S2Port : in out bool in [ 0 , 0 ] ,
```

```
v1S2Port : in out bool in [ 0 , 0 ] , v3S2Port : in out bool in [ 0 , 0 ]
par * in
        -> InputGlue [writeInputs , readOutputs , v1S0Port ,v2S1Port ,
                      v1S1Port , v2S0Port , v2S2Port , v1S2Port , v3S2Port ]
        ||
        -> Input [ v1S0Port ]
        ||
        -> Input [ v2S1Port ]
        ||
        -> Input [ v1S1Port ]
        ||
        -> Input [ v2S0Port ]
        ||
        -> Input [ v2S2Port ]
        ||
        -> Input [ v1S2Port ]
        ||
        -> Input [ v3S2Port ]
end
```

Then, the *Outputs* component allows to instantiate the *Output* process for each output LD variable.

```
component Outputs [ readOutputs : in arrayOut ] is
par * in
        -> Output [ readOutputs ] ( 0 )
        ||
        -> Output [ readOutputs ] ( 1 )
        ||
        -> Output [ readOutputs ] ( 2 )
        ||
        -> Output [ readOutputs ] ( 3 )
        ||
        -> Output [readOutputs ] ( 4 )
end
```

Both *Inputs* and *Outputs* components are instantiated in the generic *Plant* component.

```
component Plant [ writeInputs : out arrayIn ,readOutputs : in arrayOut ] is
par * in
        -> Inputs [ writeInputs ,readOutputs ]
        ||
        -> Outputs [ readOutputs ]
end
```

Finally, the main component, named *Elevation*, is defined. It instantiates the *Plant* and *PLC* components. The complete FIACRE specification is automatically generated. The specific part of the *Plant* component with the *OutputGlue* process are added manually by a composer module.

```
component Elevation is
port portInputs : in out arrayIn in [ 0 , 0 ] , portOutputs : in out arrayOut in [0 , 0 ]
par * in
        -> PLC [ portInputs , portOutputs ]
        ||
        -> Plant [ portInputs, portOutputs ] end
```

In our work, we are interested only in the generic property part and how to specify and verify it.

8.3 Application of the integration of the hidden verification activity for LD diagram

In this section, we apply our contribution on the integration of the hidden formal verification activity for LD diagrams. As shown in chapter 6, several additional elements should be added to extend the verification framework for the LD language. First, we introduce the required behavioral extensions on the LD metamodel based on the extended version of the *Executable DSML pattern*. Then, we propose the formalization of the behavioral properties and their related queries on the LD metamodel to generate their corresponding formal properties. Finally, we define a FEVEREL model based on the translational semantics of LD metamodel into FIACRE language in order to feedback verification results generated at the FIACRE level to the LD level.

8.3.1 Expressing behavioral properties

The first step for the LD expert consists in specifying the behavioral properties and their non-primitive queries. Listing 8.1 shows a possible implementation of these elements using our TOCL editor. First, the LD expert defines the signature of *isTrue()* (lines 5-7) and *isFalse()* (lines 9-11) queries. However, their bodies cannot be implemented yet because they are related to the specification of the translational semantics of LD into the semantics domain. They will be completed by the LD designer. The *isStable()* query (lines 13-15) can be defined because it is a non-primitive query.

Once the different queries are defined, he can express the behavioral property named *FreeRaceCondition* (lines 19-30).

Now, the LD designer in collaboration with the FIACRE expert and the LD expert should define the translational semantics of the LD language. This semantics should highlight the required elements to complete the specification of the primitive queries bodies. We consider the translational semantics shown in section 8.2 which maps the LD concepts into the FIACRE semantics domain.

Once the translational semantics is defined, the LD designer can complete the specification of LD primitive queries. These queries should be implemented based on the target properties language which is FIACRE properties.

Listing 8.2 shows a possible specification of LD primitive queries. Let's explain this specification for both LD queries. If a LD variable is an input or an output, it has the true (respectively false) value when its corresponding instance (FIACRE *Input* instance in the FIACRE *Inputs* component for a LD input variable and FIACRE *Output* instance in the FIACRE *Outputs* component for a LD output variable) is in the *varTrue* (respectively *varFalse*) state. Otherwise, if it is a memory variable, it has the true (respectively false) value when its corresponding boolean variable in the *Scan* process has a true (respectively false) value. This specification should be extended with the required elements to complete the generation of the FIACRE properties.

```
1 module LadderQDMM;
2 import 'http://newladder/' as ladder
```

```
3
4  // Ladder queries
5  context ladder!Variable  def : isTrue(): String=
6          //abstract query
7          ;
8
9  context ladder!Variable  def : isFalse(): String=
10         //abstract query
11         ;
12
13 context ladder!Variable  def : isStable(): String=
14   always self.isTrue() or always self.isFalse()
15         ;
16
17 // Ladder behavioral properties
18
19 context ladder!Variable
20 inv FreeRaceCondition:
21         if (self.inOutKind <> #input)
22         then
23             always
24               (ladder!Variable.allInstances()->select(v|v.inOutKind=#input)
25               ->forAll(var_input|var_input.isStable())
26             implies
27               eventually self.isStable())
28         else
29             true
30         endif
```

Listing 8.1 — LD behavioral properties and their non-primitive queries

```
1  context ladder!Variable  def : isTrue(): String=
2  if (self.inOutKind <> #mem)
3  then
4          self.getInstancePath()+ 'state varTrue'
5  else
6          thisModule.getScanPath()+'value ' + self.name
7  endif
8          ;
9
10 context ladder!Variable  def : isFalse(): String=
11 if (self.inOutKind <> #mem)
12 then
13         self.getInstancePath()+  'state varFalse'
14 else
15         thisModule.getScanPath()+ 'value (' + not self.name + ')'
16 endif;
```

Listing 8.2 — Formalization of LD primitive queries

Listing 8.3 shows additional OCL definitions defined to produce the corresponding *path* for each observable element in the FIACRE side. These definitions should take into account the defined translational semantics and the hierarchical structure to describe the composition of components and processes in FIACRE.

```
1  context ladder!Variable def: getindex() : Integer =
2  if self.inOutKind = #input then
3          ladder!Variable.allInstances()->select(v |v.inOutKind = #input)->indexOf(self)
4  else
5                  ladder!Variable.allInstances()->select(v | v.inOutKind <> #input)->indexOf(self)
6  endif;
7
8  context ladder!Variable def: getVariableInstance() : Integer =
```

```
if self.inOutKind = #input then
        self.getindex() + 1
else
        self.getindex()
endif ;

def: getMain(): String=
        ladder!Program.allInstances()->first().name+'/'
        ;


def: getPLCInstanceInMain(): String=
        '1/'
        ;
def: getScanInstanceInPLC(): String=
        '1/'
        ;
def: getScanPath(): String=
        thisModule.getMain()+thisModule.getPLCInstanceInMain()+thisModule.getScanInstanceInPLC()
        ;

def: getPlantInstanceInMain(): String=
        '2/'
        ;
def: getOutputsInstanceInPlant(): String=
        '2/'
        ;
def: getInputsInstanceInPlant(): String=
        '1/'
        ;
def: getInputsPath(): String=
        thisModule.getMain()+thisModule.getPlantInstanceInMain()+thisModule.getInputsInstanceInPlant()
        ;

def: getOutputsPath(): String=
        thisModule.getMain()+thisModule.getPlantInstanceInMain()+thisModule.getOutputsInstanceInPlant()
        ;

context ladder!Variable  def : getInstancePath(): String=
if self.inOutKind = #input
then
        thisModule.getInputsPath()
else
        thisModule.getOutputsPath()
endif
+ self.getVariableInstance()
+'/'
        ;
```

Listing 8.3 — Completing the TOCL specification for LD

Once different elements related to the specification of behavioral properties are defined, the LD designer can proceed to generate the corresponding ones at the FIACRE level. As explained in the chapter 4, The complete TOCL specification is transformed into an ATL query which takes a LD model as input and generates the corresponding FIACRE properties.

```
property FreeRaceCondition_8 is ltl (
 [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
     and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
     and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
     and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
     and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
     and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
```

```
8       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
9       => <> ( ( [] ( Elevation/2/2/1/state varTrue ) or [] ( Elevation/2/2/1/state varFalse ) ) )
10    ) ) )
11  assert FreeRaceCondition_8
12  property FreeRaceCondition_9 is ltl (
13   [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
14       and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
15       and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
16       and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
17       and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
18       and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
19       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
20       => <> ( ( [] ( Elevation/2/2/2/state varTrue ) or [] ( Elevation/2/2/2/state varFalse ) ) )
21    ) ) )
22  assert FreeRaceCondition_9
23  property FreeRaceCondition_10 is ltl (
24   [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
25       and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
26       and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
27       and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
28       and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
29       and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
30       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
31       => <> ( ( [] ( Elevation/2/2/3/state varTrue ) or [] ( Elevation/2/2/3/state varFalse ) ) )
32       ) ) )
33  assert FreeRaceCondition_10
34  property FreeRaceCondition_11 is ltl (
35   [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
36       and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
37       and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
38       and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
39       and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
40       and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
41       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
42       => <> ( ( [] ( Elevation/2/2/4/state varTrue ) or [] ( Elevation/2/2/4/state varFalse ) ) )
43       ) ) )
44  assert FreeRaceCondition_11
45  property FreeRaceCondition_12 is ltl (
46   [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
47       and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
48       and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
49       and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
50       and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
51       and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
52       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
53       => <> ( ( [] ( Elevation/2/2/5/state varTrue ) or [] ( Elevation/2/2/5/state varFalse ) ) )
54       ) ) )
55  assert FreeRaceCondition_12
56  property FreeRaceCondition_13 is ltl (
57   [] ( ( ( [] ( Elevation/2/1/2/state varTrue ) or [] ( Elevation/2/1/2/state varFalse ) )
58       and ( [] ( Elevation/2/1/3/state varTrue ) or [] ( Elevation/2/1/3/state varFalse ) )
59       and ( [] ( Elevation/2/1/4/state varTrue ) or [] ( Elevation/2/1/4/state varFalse ) )
60       and ( [] ( Elevation/2/1/5/state varTrue ) or [] ( Elevation/2/1/5/state varFalse ) )
61       and ( [] ( Elevation/2/1/6/state varTrue ) or [] ( Elevation/2/1/6/state varFalse ) )
62       and ( [] ( Elevation/2/1/7/state varTrue ) or [] ( Elevation/2/1/7/state varFalse ) )
63       and ( [] ( Elevation/2/1/8/state varTrue ) or [] ( Elevation/2/1/8/state varFalse ) )
64       => <> ( ( [] ( Elevation/1/1/value Y6 ) or [] ( Elevation/1/1/value ( not ( Y6 ) ) ) ) )
65       ) ) )
66  assert FreeRaceCondition_13
```

Listing 8.4 — The generated FIACRE properties for the elevation system

Listing 8.4 shows the FIACRE properties corresponding to the LD elevation system given in Figure 8.6.

Once the FIACRE model corresponding to a LD model and its related properties are generated, the formal verification can be performed. The full FIACRE specification is translated by the FRAC compiler into a TTS specification, the accepted input of the TINA toolbox. The

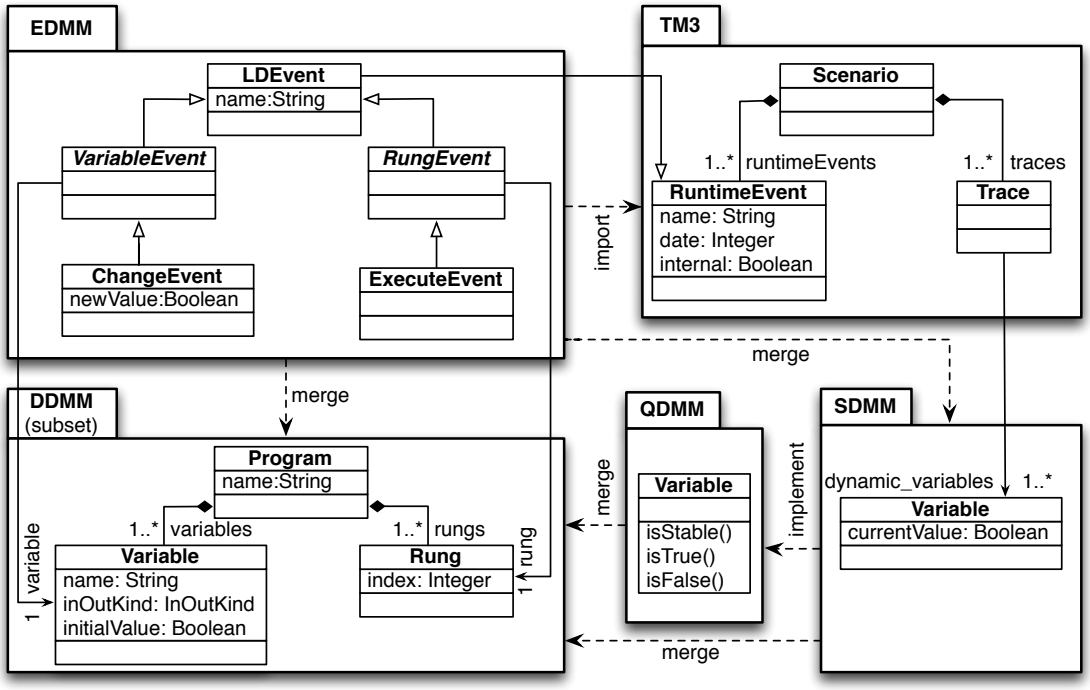

Figure 8.8 — The executable LD (xLD)

verification of the FIACRE program shows that the LD model shown in Figure 8.6 is free of race condition.

8.3.2 Introducing behavioral extensions

In order to integrate the complete verification activity for LD, the LD designer should define different behavioral extensions for its domain even before implementing the translational semantics.

It consists in introducing different behavioral extensions related to the verification and the execution of a LD programs. Figure 8.8 shows a possible implementation of an executable LD metamodel (xLD). The DDMM defines the abstract syntax of LD domain. The SDMM defines an additional attribute to record the current value of a LD variable (*currentValue*) during the execution of a LD program. Therefore, for the QDMM, two queries may be identified on *Variable* metaclass : *isFalse()* and *isTrue()*. In order to verify generic behavioral properties for LD, an additional query, *isStable()*, is defined to ask whether a LD variable is stable or not. To show how a LD model evolves, two LD events are identified: execute a LD rung (*ExecuteEvent*) and change the value of a LD variable (*ChangeEvent*). Finally, the TM3 is the same as the one presented for XSPEM and TPN, as it is DSML-independent.

8.3.3 Feedback verification results

The last work for the LD designer during the integration of the hidden formal verification for its domain consists in feeding back verification results generated in the formal level. In chapter 7, we have shown how we feedback verification results from the TPN level to the FIACRE one. The LD designer now should complete the second step by defining the appropriate mappings between LD events (defined in the LD EDMM) and their corresponding ones in the FIACRE level using FEVEREL.

```
1  import "http://ladderSemantics/1.0" as LadderSemantics
2  import "http://fiacreSemantics/1.0" as FormalSemantics
3  import "http://newladder/1.0" as Ladder
4  import "http://www.topcased.org/fiacre/xtext/Fiacre" as Formal
5
6  events mapping EnterEvent2ChangeEvent:
7    DSMLEvent ch_e: LadderSemantics.ChangeEvent
8      (value <- if en_e.state.name = 'varFalse' then false else true endif)
9    maps
10   FormalEvent en_e :FormalSemantics.EnterEvent (
11         Formal!ComponentDeclaration.allInstances()->select(c|c.name= 'Outputs')
12         ->first().body.blocks->indexOf(en_e.path.instances->last().instance)
13         =
14         Ladder!Program.allInstances()->first().variables->select(c|c.inOutKind= #output)
15         ->indexOf(ch_e.variable)
16         and
17         Formal!ComponentDeclaration.allInstances()->select(c|c.name= 'Outputs')
18         ->first().body.blocks->indexOf(en_e.path.instances->last().instance) > 0
19 )
20 end events mapping
21
22 events mapping ChangeEvent2ExecuteEvent:
23   DSMLEvent ex_e: LadderSemantics.ExecuteEvent
24   maps
25   FormalEvent ch_e :FormalSemantics.ExitEvent (
26     if (ch_e.state.name.startsWith('rung')) then
27         ch_e.state.name.substring(6, 6).toInteger()
28         =
29         Ladder!Program.allInstances()->first().rungs->indexOf(ex_e.rung)
30       else
31         false
32       endif
33 )
34 end events mapping
```

Listing 8.5 — The definition of events mappings for the verification of LD models using FIACRE

Listing 8.5 shows a possible specification of these mappings. The first one consists in specifying the *ChangeEvent* of a LD variable. This event is triggered when the corresponding FIACRE process instance (*Output* process instance in the *Outputs* component for output LD variables) changes its state. The mapping consists in verifying whether the index of the LD variable corresponds to the index of the FIACRE instance in the FIACRE *EnterEvent*. The *value* attribute of the generated LD *ChangeEvent* is initialized based on the state of the FIACRE *EnterEvent* (line 8). The second mapping concerns capturing the execution of a LD rung *ExecuteEvent*. It is observed when the *Scan* process instance enters in the corresponding state. Entering in the state *rung_i* corresponds to the i^{th} rung in the LD program.

Once the mappings are defined, the feedback of verification results can be performed.

For example, in the LD model shown in Figure 8.3, the *E* output variable does not stabilize. Listing 8.6 shows a subset of the generated counter-example.

```
1  ExecuteEvent{1}
2  ExecuteEvent{2}
3  ExecuteEvent{3}
4  ExecuteEvent{4}
5  ChangeEvent{E, true}
6  ExecuteEvent{1}
7  ExecuteEvent{2}
8  ExecuteEvent{3}
9  ExecuteEvent{4}
10 ChangeEvent{E, false}
11 ExecuteEvent{1}
12 ExecuteEvent{2}
13 ExecuteEvent{3}
14 ExecuteEvent{4}
15 ChangeEvent{E, true}
```

Listing 8.6 — The definition of events mappings for the verification of LD models using FIACRE

8.4 Conclusion

In this chapter, we have proposed a possible implementation of a verification toolchain for LD language. This toolchain allows to provide a seamless approach to verify whether LD programs behave as expected while hiding all formal aspects for industrial users. Using our TOCL proposal and its tooling, we have successfully implemented different required behavioral properties in LD level and then translated them into the formal side. This generation of formal properties is guided by the proposed translational semantics and thus by the proposed queries. Next, using our FEVEREL proposal and its tooling, we define the required mappings to feedback verification results from the formal side into the LD one.

Conclusion

Résumé

Cette conclusion résume les résultats obtenus durant cette thèse et présente des perspectives de ce travail.

Le défi relevé par cette thèse était d'intégrer les activités de vérification formelle dans le processus de développement des DSMLs. L'approche adoptée est basée sur la définition d'une sémantique translationnelle d'un DSML vers un domaine sémantique formel afin de réutiliser les outils puissants (simulateur ou vérificateur par exploration exhaustive des modèles) disponibles dans ce domaine.

Notre contribution consiste à faciliter l'intégration de l'activité de vérification formelle dans l'outillage d'un DSML tout en gardant tous les aspects formels cachés pour les utilisateurs finaux. Cette caractéristique est un élément clé pour le succès de l'utilisation de la vérification formelle par exploration exhaustive des modèles par des non experts. En outre, les modèles formels de vérification sont produits en s'appuyant sur des modèles formels intermédiaires qui sont eux-mêmes générés à partir des modèles conformes à un DSML. La figure 8.9 montre l'architecture d'un vérificateur d'un DSML comme proposé dans [VDW12]. Le concepteur d'un DSML se concentre sur les préoccupations de son DSML et les traductions nécessaires vers le niveau intermédiaire (la partie rouge). Ainsi, pour l'expert du DSML et l'utilisateur final ne doivent se concentrer que sur leur domaine sans avoir à se préoccuper d'un autre domaine, formel ou IDM.

Connecter deux domaines différents, les méthodes formelles et l'IDM, était encore un défi quand l'objectif est de fournir une approche générique et non spécifique à un seul DSML. Le principal intérêt de notre approche est de permettre l'utilisation des outils puissants de la vérification de modèle pour évaluer les modèles au plus tôt dans le processus de développement. Notre approche fournit des outils appropriés pour intégrer facilement la vérification comportementale pour les DSMLs.

Notre contribution consiste également à cacher aux utilisateurs finaux des DSMLs cette sémantique translationnelle ainsi que d'autres aspects formels liés au domaine formel ciblé. Ce travail est basé sur un patron de métamodélisation (*Executable DSML pattern*) qui favorise la définition d'outils génératifs, et facilite ainsi l'intégration des outils pour de nouveaux DSMLs.

Nous avons suivi une approche dirigée par les cas d'utilisation comme c'est fait couramment en IDM. Nous avons sélectionné un cas d'utilisation approprié qui permet de valider

notre proposition. Nous avons développé des outils ad-hoc ciblant la généricité. Nous avons proposé des transformations d'ordre supérieur (HOT) pour produire les outils qui ont été prototypés manuellement. Nos principales contributions sont les suivantes:

- Dans le chapitre 4, nous fournissons, pour les différents acteurs concernés, un langage approprié (TOCL), sous forme d'une extension temporelle du langage OCL, proche de leurs connaissances en IDM pour définir les différents éléments nécessaires pour exprimer des propriétés comportementales au niveau DSML. Nous proposons une extension supplémentaire au patron de métamodélisation, nommé le métamodèle de définition de requête (QDMM), afin de spécifier les différentes requêtes possibles qui peuvent être demandées à un modèle conforme à un DSML durant son exécution. L'éditeur TOCL permet à l'expert du DSML de définir à la fois les propriétés comportementales attendues et les requêtes du DSML.

- Le deuxième problème abordé dans cette thèse est la remontée des résultats de vérification qui sont obtenus au niveau formel générés par les outils d'exploration exhaustive des modèles. Ils ont besoin d'être mis à profit à l'utilisateur final pour les comprendre, même s'il n'a pas de compétences en méthodes formelles. Dans le chapitre 5, nous proposons une approche basée sur les modèles pour faciliter la remontée des résultats de vérification pour le concepteur d'un DSML. Ce processus est basé sur les extensions comportementales définies lors de l'application du patron de métamodélisation (*Executable DSML pattern*) à la fois au DSML et au langage formel choisi. Après avoir écrit manuellement la transformation de retour et expérimenté les transformations de modèle bidirectionnelles, nous avons proposé une solution plus adaptée à l'utilisateur: un langage dédié pour spécifier la remontée des informations. Ce langage, nommé FEVEREL (Feedback Verification Results Language en anglais), permet de définir des correspondances entre les événements du DSML et les événements du niveau formel. Le but de ce langage consiste à utiliser OCL pour spécifier comment un événement du niveau DSML peut être observé au niveau formel. Ensuite, notre approche s'appuie sur l'utilisation d'une transformation d'ordre supérieur qui accepte un modèle FEVEREL et génère une deuxième transformation de modèle. Cette dernière produit un scénario au niveau DSML à partir des résultats de la vérification générés dans le niveau formel.

- Nous avons proposé une méthode pour construire un framework de vérification pour un nouveau DSML. Nous avons expliqué les étapes à réaliser par les différents acteurs concernés pour produire un framework de vérification pour un DSML. En outre, nous avons montré les différents scénarios possibles pour adapter un framework de vérification lors du changement de certaines de ses composantes. Enfin, nous avons fourni plusieurs lignes directrices pour la validation de l'ensemble des outils de vérification en utilisant notre proposition de TOCL.

- Un autre problème qui doit être abordé est le fossé sémantique entre les DSMLs et les langages formels qui peut rendre difficile la définition de la sémantique translationnelle. Ainsi, nous avons adapté la chaîne d'outils de vérification afin d'intégrer le langage intermédiaire FIACRE. Intégrer FIACRE comme un langage formel nécessite étendre sa syntaxe abstraite avec les événements (dans le EDMM) et les états (dans le SDMM) qui

peuvent être capturés lors de l'exécution. Puis, en utilisant FIACRE comme un nouveau langage formel cible permet de valider notre approche.

Les travaux de cette thèse visent un problème essentiel : l'intégration des méthodes formelles d'une manière transparente en IDM sous forme des outils de V&V pour les DSMLs. Ils ouvrent plusieurs perspectives :

- **Intégrer d'autres langages de propriétés dans l'éditeur TOCL**: Notre éditeur TOCL et son outillage prennent actuellement en considération la logique LTL pour spécifier les propriétés comportementales et leur génération en propriétés formelles. Toutefois, cette extension temporelle limite les experts d'un DSML et les concepteurs à modéliser leurs propriétés comportementales uniquement avec les opérateurs de LTL. Pour lever cette limite, l'éditeur TOCL pourrait être étendu afin d'accepter des logiques formelles supplémentaires comme CTL [EC82] et les patrons de Dwyer [DAC98]. En outre, il serait utile d'intégrer plusieurs langages de propriété dans notre approche en proposant une approche générique (XOCL2X) qui donne les spécifications et les outils nécessaires pour étendre le langage OCL et supporter d'autres langages (étiquetés X). Dans cet esprit, il sera intéressant de fournir des langages de propriété plus proches des utilisateurs finaux des DSMLs qui s'appuient sur des modèles de propriétés ressemblant aux langages naturels.

- **Étendre la remontée des résultats de vérification**: Actuellement, la remontée des résultats de vérification se concentre sur les événements du DSML (instances de l'EDMM du DSML). Elle consiste à définir les correspondances entre les événements du DSML et les événements correspondants au niveau formel en utilisant le langage FEVEREL. Les résultats de la vérification sont générés sous forme d'une succession d'événements (un scénario). FEVEREL pourra être étendu pour remonter également les états, et l'intégralité de trace d'exécution (les événements et des états) générés par le model-checker.

  De plus, à la remontée des événements, il est parfois nécessaire de recourir aux états précédents et suivants pour spécifier la correspondance des événements appropriés parce que les événements générés dans le côté formel peuvent ne pas contenir toutes les informations nécessaires à la remontée au niveau DSML. En outre, il est obligatoire d'étendre le langage FEVEREL pour supporter des correspondances plus sophistiquées comme la correspondance 1-à-n et la correspondance m-à-n. Pour adapter le langage FEVEREL à cette approche, il devrait être nécessaire de choisir le patron approprié pour capturer les événements. Complex Event Processing (CEP) [BK09] sera un candidat intéressant pour le faire.

- **Compléter l'approche de processus de construction le framework de vérification pour un nouveau DSML**: Enfin, il serait utile pour l'expert du DSML et le concepteur de fournir un outil pour combiner les différents éléments liés à l'activité de vérification (EDMM, SDMM et QDMM) qui peuvent être montrés comme une spécification complète qui est utilisée pour (1) spécifier la sémantique translationnelle à travers les relations entre ces différents éléments, (2) la valider en définissant des conditions de pré-post qui peuvent être évaluées après la mise en oeuvre, (3) mettre en œuvre les

propriétés comportementales et (4) remonter des résultats de la vérification. Il peut être intéressant de générer à partir de cette spécification un squelette de la sémantique translationnelle, la requête de génération des propriétés comportementales au niveau formel, et la transformation de retour des résultats de vérification.

THIS conclusion summarizes the results obtained during this PhD and presents perspectives for this work.

Our approach helps in integrating formal verification in the DSML tooling while keeping all formal aspects hidden to the end-users. This characteristic is a key feature for the success of the use of model checking techniques by non expert. Furthermore, the formal verification models are produced relying on intermediate formal models that are generated from the DSML models. Figure 8.9 shows the architecture of a DSML model-checker as proposed in [VDW12]. The DSML designer focuses on the DSML concerns and their mappings into the intermediate level (the red part). Thus, both the DSML expert and the DSML end-user only need to focus on their domain without dealing with the formal domain or the MDE domain.

Connecting two different domains, formal methods and MDE, was still a challenge especially when we target providing a generic approach and not an ad-hoc one that focuses on one specific formal domain to assess models designed in one specific DSML.

The most important advantage of our approach is the use of powerful tools to assess models as early as possible in the development process. Our approach provides appropriate tools to integrate easily the behavioral verification for DSMLs while hiding all formal aspects to end-users.

Main contributions and discussions

The main goal of this thesis was to integrate formal verification activities in the development process of DSMLs. The adopted approach is based on the definition of a translational semantics from the DSML to a formal semantic domain in order to reuse the powerful tools (simulator or model-checker) available in this domain.

Our contribution consists in hiding to the DSML end-users this translational semantics as well as other formal aspects related to the targeted formal domain. This work is based on a metamodeling pattern for executable DSML (*Executable DSML pattern*) that favors the definition of generative tools proposed by Combemale et al. and slightly extended in this work, and thus eases the integration of tools for new DSMLs.

We have followed a use case driven approach as commonly done in MDE. We have selected an appropriate use case that allows to validate the proposal. We have developed ad-hoc tools targeting genericity. We have proposed higher-order transformations (HOTs) to produce the tools that were manually prototyped. Our main contributions are as follows.

Expressing DSML behavioral properties

DSML experts and end-users must be able to define the properties they want to assess on their models. In chapter 4, we provide, for the different actors concerned by the use of a DSML, a suitable language (TOCL), as a temporal extension of the OCL standard, close to their knowledge in the MDE domain to define different required elements to express behavioral properties at the DSML level. We propose an additional extension to the *Executable DSML pattern*, named Query Definition MetaModel (QDMM), to specify different possible queries that can be asked on a DSML conforming model during its execution. The TOCL editor allows the DSML expert to define both expected behavioral properties and DSML

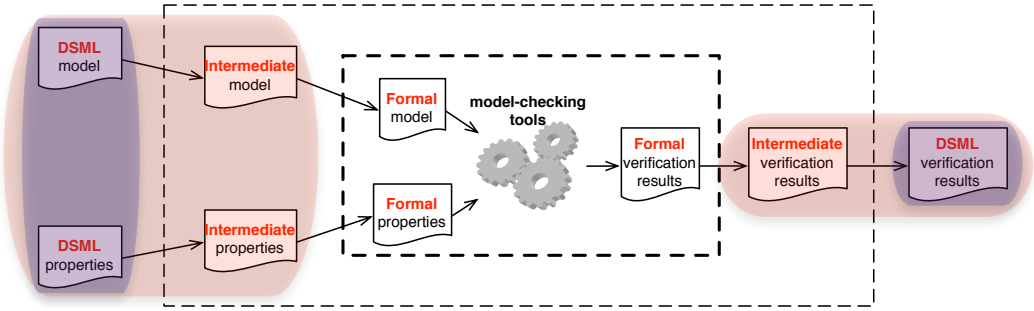

Figure 8.9 — The architecture of a domain-specific model checker

queries. These queries are either non-primitive and can be directly implemented; or primitive that can only be specified. Then, the DSML designer defines these primitive queries based on the implemented translational semantics. This tool can also be used to specify properties targeting specific models that are assessed at the DSML end-user request. In addition, we provide the necessary tooling to generate the corresponding formal properties. The tooling is provided as a higher-order transformation that generates at first an ATL query from the TOCL specification. This generated query accepts a DSML conforming model and generates the corresponding formal properties which are, along with the formal models (generated by running the translational semantics), the input elements of the model-checker that performs the formal verification.

To obtain this generic tool, we have started by manually writing these properties in the formal side. Then, in order to automatically generate them, we have written an ATL query that accepts a DSML conforming model. Finally, as we aimed to obtain a suitable tool to express behavioral properties, we developed the TOCL tool to automatically generate the ATL query. We extended the *Executable DSML pattern* to support the "query" notion.

In our approach, expressing behavioral properties at the DSML level with the TOCL editor is still partial because it only allows the use of the linear temporal logic as we only provide the tooling that targets the LTL language in the formal side. The DSML expert may need to target other temporal logics. The current tooling is parameterized in order to be able to target other temporal languages. As the TOCL language is still low-level for some DSML end-users that can have difficulties with OCL, it might be needed to define a domain-specific language for the properties.

Feedback verification results on the DSML level

The second problem addressed in this PhD is the feedback of verification results which are obtained in the formal side after performing the verification by model-checking tools. They need to be leveraged in order for the end-user to understand them even if he has no skills in formal methods. In chapter 5, we propose a model-based approach to ease the feedback of the verification results for the DSML designer. This process is based on the runtime extensions defined during the application of the *Executable DSML pattern* both at the DSML and the formal levels. At first, we focus either on encoding manually the backward model

transformation or on providing a bidirectional model transformation that defines both the translational semantics and the feedback verification results. These solutions are ad-hoc for the DSML designer and do not fully address his needs. Then, we propose a DSPL, named FEVEREL, using Xtext to define mappings between formal events and DSML events. The purpose of this language consists in using OCL to specify how a DSML event can be observed at the formal level. Then, our approach relies on a generic higher-order transformation that accepts a FEVEREL mapping and generates a model-to-model transformation. This latter produces a DSML scenario from the verification results generated in the formal side.

Our work follows an example driven approach to finally obtain the FEVEREL proposal. At first, we started by studying how we obtain verification results in DSML level. The first experiments were specific to the implementation of the translational semantics and also to the DSML abstract syntax. As our experiments progressed in [ZCP13b, ZCP13a], we became aware that mapping information was required. The mapping specified using FEVEREL can generate only one DSML event (1-to-1 mapping) or a set a DSML events (1-to-n mapping) that are instances of the same DSML event meta-class. However, it fails so far to define a general 1-to-n mappings or m-to-n mappings. In addition, sometimes the mappings between events are not sufficient to feedback verification results. Additional information related to the state of the formal model before and after triggering the formal event can be required to define a full mapping that eases the feedback of verification results. It was possible to add two references between the `RuntimeEvent` and `Trace` meta-classes in the TM3 (each event has a *previous* and a *next* trace showing the current state of the model during execution) to access this information.

Process to build a verification framework for a new DSML

Based on both proposed contributions, in chapter 6, we focused on the methodological side to build a verification framework for a new DSML. We have explained the performed steps by the different concerned actors to produce a verification framework for a DSML. In addition, we have showed the different possible scenarios to adapt the verification framework when some of its components change. Finally, we have provided several guidelines for validating the verification toolchain using our TOCL proposal.

This process targets the definition of new verification toolchain. It could be adapted if the transformation is already available. However, it does not fit if the transformation is a black-box, which was experimented in the following part.

Introducing an intermediate language in the verification toolchain

Another problem that must be tackled is the semantic gap between the DSML and formal languages which may make difficult the definition of the translational semantics. Thus, we have adapted the verification toolchain to integrate the FIACRE intermediate language. Integrating FIACRE as a formal language requires extending its abstract syntax with the events (in the EDMM) and the states (in the SDMM) that are of interest during the runtime execution. Then, using FIACRE as a new target formal language allows to validate our approach — the architecture of the verification toolchain and method to build or update it.

This integration eases the implementation of the translational semantics for the DSML

designers. It can be considered as a transformation from a DSML to another formal DSML (DSMLToDSML). We have identified the various elements that should be updated in the verification toolchain conforming to the study proposed in chapter 6: the primitive queries and the events mappings that are both related to the translational semantics.

Perspectives

The works in this PhD targeted a key problem: the integration of hidden formal methods in MDE as V&V tools for DSMLs. We advocate that our approach allows to ease the building of the required tools to specify easily the expected behavioral properties on DSML conforming models and then to obtain the verification results in the DSML domain.

Integrating additional languages in the TOCL editor

Our TOCL editor and its tooling supports the LTL operators to specify DSML behavioral properties and then to generate the corresponding formal properties. However, this temporal extension limits the DSML experts and the designers to model their behavioral properties only with LTL operators. A possible way to avoid this is to extend the TOCL editor to support additional formal language like CTL [EC82] and Dwyer patterns [DAC98]. This extension would allow the DSML expert to verify more sophisticated behavioral properties. It would require extending the TOCL grammar and the higher-order transformation TOCL2ATL to support these new extensions. In addition, it would be helpful to integrate more property languages in our approach by proposing a generic approach (XOCL2X) that gives the mandatory specifications and tools to extend the OCL language and support other languages (tagged X). In this spirit, it will be interesting to provide property languages closer to the DSML end-users that rely on property patterns looking like natural language (so called boilerplate requirements). This specification can be mapped into an eventual XOCL tool.

Extending the feedback of verification results

The current feedback of verification results focus on DSML events (instances of DSML EDMM). It consists in defining mappings between DSML events and their corresponding formal ones using our FEVEREL proposal. The generated verification results are shown as a succession of events (a scenario). This current implementation does not favor the feedback of the execution trace (both events and states) generated by the model-checker. It is necessary to extend the FEVEREL language to support the feedback of states. In addition, to feedback the DSML events, it is required sometimes to resort to the previous and next states to specify the appropriate events mapping because events generated in the formal side may not contain all required information to feedback it in the DSML side. Model checkers may not build all the data in the states as it relies on the abstraction to reduce the size of the state space. Other tools can provide these data by running the scenario in a simulator at the formal or user model level. In addition, it is mandatory to extend the FEVEREL language to support more sophisticated mappings like full 1-to-n and m-to-n mappings. To adapt the FEVEREL language to this approach, it should be necessary to choose the appropriate pattern to capture events. Complex Event Processing (CEP) [BK09] should be an interesting candidate to do that.

Complete the process approach to build a verification framework for a new DSML
Finally, it would be helpful for the DSML expert and designer to provide a tool to combine the different fragments related to verification activity (EDMM, SDMM and QDMM) that can be shown as a full specification which is used to (1) specify the translational semantics with relations between these elements, (2) validate it by defining pre-post conditions that can assessed after the implementation, (3) implement the behavioral properties and (4) feedback the verification results. It can be interesting to generate from this specification a "pseudo" implementation of the translational semantics, the query to generate the behavioral properties in the formal level, and the transformation to feedback the verification results.

Part

Appendices

A Related publications

International conferences articles

- Faiez Zalila , Xavier Crégut and Marc Pantel. A user-oriented approach to integrate formal verification activity for DSML (regular paper). In : European Congress on Embedded Real Time Software and Systems (ERTS2, 2014), Toulouse, 05/02/2014-09/02/2014, 2014.
- Faiez Zalila, Xavier Crégut, Marc Pantel. A transformation-driven approach to automate feedback verification results (regular paper). Dans : International Conference On Model and Data Engineering (MEDI 2013), Amantea, Calabria, Italy, 25/09/13-27/09/13, Springer, Lecture Notes in Computer Science, p. 266-277, 2013.
- Faiez Zalila, Xavier Crégut, Marc Pantel. Formal Verification Integration Approach for DSML (regular paper). Dans : ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, 29/09/13-04/10/13, Springer-Verlag, septembre 2013.
- Faiez Zalila, Xavier Crégut, Marc Pantel. Leveraging formal verification tools for DSML users: a process modeling case study (regular paper). Dans : ISoLA Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Amirandes, Heraclion, Crete, 15/10/12-18/10/12, Vol. 7610, Springer, Lecture Notes in Computer Science, p. 329-343, 2012.
- Benoit Combemale, Xavier Crégut, Arnaud Dieumegard, Marc Pantel, Faiez Zalila. MDE through the Formal Verification of Process Models (regular paper). Dans : Educators' Symposium@MODELS 2011 - Software Modeling in Education, Wellington, New Zealand, 18/10/11, Vol. 52, Electronic Communications of the EASST, (en ligne), 2012.

National conferences articles

- Faiez Zalila, Xavier Crégut, Marc Pantel. Verification results feedback for FIACRE intermediate language (student paper). Dans : Conférence en Ingénierie du Logiciel (CIEL 2012), Rennes, France, 19/06/12-21/06/12, IRISA, 2012.
- Faiez Zalila, Xavier Crégut, Marc Pantel. Approche transparente pour la vérification des modèles métiers Journées sur l'Ingénierie Dirigée par les Modèles, IDM-2011, Lille, France, Juin 2011.

Bibliography

[Abi12] Nouha Abid. *Verification of Real Time Properties in Fiacre Language*. Thèse de doctorat, LAAS-CNRS, Toulouse, France, Décembre 2012.

[AFS98] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. In *Proceedings of the 1st Conference on Tools and Algorithms for the Analysis and Construction of Systems*, March 1998.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Patrick Farail, Pierre Gaufillet, Hubert Garavel, and Frederic Lang. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In 4^{th} *European Congress ERTS Embedded Real-Time Software (2008)*, January 2008.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In *Proceedings of the 9^{th} International Conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 440–453, Berlin, Heidelberg, 2006. Springer-Verlag.

[BCC+08] Darlam Fabio Bender, Benoît Combemale, Xavier Crégut, Jean-Marie Farines, Bernard Berthomieu, and François Vernadat. Ladder Metamodeling and PLC Program Validation through Time Petri Nets. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Berlin, Germany, 09/06/2008-12/06/2008*, volume 5095 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

[BCL+01] Andrea Bondavalli, Mario Dal Cin, Diego Latella, István Majzik, András Pataricza, and Giancarlo Savoia. Dependability Analysis in the Early Phases of UML Based System Design. *Journal of Computer Systems Science and Engineering*, 16:265 – 275, 2001.

[BCMP12] Jean-Christophe Bach, Xavier Crégut, Pierre-Etienne Moreau, and Marc Pantel. Model transformations with tom. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March* 31^{st} *- April* 1^{st}, 2012.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1^{st} edition, 2012.

[Ben08] Darlam Fabio Bender. Application of the MDE for Modeling and Formal Verification of PLC Programs written in Ladder Diagram Language. DAS 5511: Projeto de Fim de Curso, Universidade Federal de Santa Catarina, 2008.

[Bey92] Martin Beyer. *AGG 1.0 – Tutorial*, 1992.

[Béz04] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[Béz06] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 36–64, Berlin, Heidelberg, 2006. Springer-Verlag.

[BFS00] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal*, 9(1):76–110, March 2000.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the* 16^{th} *IEEE International Conference on Automated Software Engineering*, ASE '01, pages 273–280, Washington, DC, USA, 2001. IEEE Computer Society.

[BGHM05] Mikael Buchholtz, Stephen Gilmore, Valentin Haenel, and Carlo Montangero. End-to-End Integrated Security and Performance Analysis on the DEGAS Choreographer Platform. In *Proceedings of the 2005 International Conference on Formal Methods*, FM'05, pages 286–301, Berlin, Heidelberg, 2005. Springer-Verlag.

[Bie10] Matthias Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010.

[BK09] Alejandro P. Buchmann and Boris Koldehofe. Complex event processing. *it - Information Technology*, 51(5):241–242, 2009.

[BRV04] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[BSE03] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional, 2003.

[BVWW09] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Model Checking Flight Control Systems: the Airbus Experience. In 31^{st} *International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 18–27, 2009.

[Car12] Ana Maria Mainhardt Carpes. Properties of LD Programs: Expression and Verification. DAS 5511: Projeto de Fim de Curso, Universidade Federal de Santa Catarina, 2012.

[CBF+10] Tiago Correa, Leandro Buss Becker, Jean-Marie Farines, Jean-Paul Bodeveix, Mamoun Filali, and François Vernadat. Supporting the Design of Safety Critical Systems Using AADL. In 15th *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 331–336, March 2010.

[CBG+08] Agusti Canals, Hugues Bonnin, Sébastien Gabel, Christophe Le Camus, and Rodrigo Saad. An OpenEmbeDD Experimentation: Transformation from an SDL Profiled UML Model to a FIACRE Model. In 4th *European Congress Embedded Real Time Software (ERTS)*, January 2008.

[CCBV07] Benoît Combemale, Xavier Crégut, Bernard Berthomieu, and François Vernadat. SimplePDL2Tina : Mise en oeuvre d'une Validation de Modèles de Processus. In Hermes Sciences/Lavoisier, editor, 3^{ieme} *journées sur l'Ingénierie Dirigée par les Modeles (IDM, in french)*, pages 86–101, Toulouse, France, 2007.

[CCG+07] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and François Vernadat. A property-driven approach to formal verification of process models. In Joaquim Filipe, José Cordeiro, and Jorge Cardoso, editors, 9th *International Conference Enterprise Information Systems (ICEIS) 2007, Funchal, Madeira, June 12-16, 2007, Revised Selected Papers*, volume 12 of *Lecture Notes in Business Information Processing*, pages 286–300. Springer, 2007.

[CCO+04] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004.

[CCP+10] Xavier Crégut, Benoît Combemale, Marc Pantel, Raphael Faudoux, and Jonatas Pavei. Generative Technologies for Model Animation in the TOPCASED Platform. In 6th *European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6138 of *LNCS*, Paris, France, June 2010. Springer.

[CCP12] Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *Proceedings of the 2012* 19th *Asia-Pacific Software Engineering Conference - Volume 01*, APSEC '12, pages 282–287, Washington, DC, USA, 2012. IEEE Computer Society.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proceedings of the* 2nd *International Conference on*

Theory and Practice of Model Transformations, ICMT '09, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[CG12] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *Proceedings of the* 12^{th} *International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM'12, pages 58–90, Berlin, Heidelberg, 2012. Springer-Verlag.

[CGCT07] Benoît Combemale, Pierre-Loic Garoche, Xavier Crégut, and Xavier Thirioux. Towards a Formal Verification of Process Model's Properties – SimplePDL and TOCL case study. In 9^{th} *International Conference on Enterprise Information Systems (ICEIS 2007)*, pages 80–89, Funchal, Madeira - Portugal, June 2007. INSTICC press.

[CGR11] Benoît Combemale, Laure Gonnord, and Vlad Rusu. A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In *Proceedings of the* 7^{th} *European Conference on Modelling Foundations and Applications*, ECMFA'11, pages 35–51, Berlin, Heidelberg, 2011. Springer-Verlag.

[CGS12] Hyun Cho, Jeff Gray, and Eugene Syriani. Creating Visual Domain-specific Modeling Languages from End-user Demonstration. In *Proceedings of the* 4^{th} *International Workshop on Modeling in Software Engineering*, MiSE '12, pages 22–28, Piscataway, NJ, USA, 2012. IEEE Press.

[CHJ+12] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *SLE2012 -* 5^{th} *International Conference on Software Language Engineering*, LNCS. Springer, September 2012.

[Com03] International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers – Part 3*. Geneva, Switzerland, 2nd edition, 2003.

[Com08] Benoît Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2008.

[CRC+06] Benoît Combemale, Sylvain Rougemaille, Xavier Crégut, Frédéric Migeon, Marc Pantel, Christine Maurel, and Bernard Coulette. Towards a Rigorous Metamodeling. In 2^{nd} *International Workshop on Model-Driven Enterprise Information Systems (MDEIS)*, Paphos, Cyprus, May 2006. INSTICC.

[DAC98] Matthew Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[DMGB09] Zekai Demirezen, Marjan Mernik, Jeff Gray, and Barrett Bryant. Verification of DSMLs Using Graph Transformation: A Case Study with Alloy. In *Proceedings of the* 6^{th} *International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, pages 3:1–3:10, New York, NY, USA, 2009. ACM.

[EC82] E.Allen Emerson and Edmund M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.

[FDQDR+11] Jean-Marie Farines, Max H. De Queiroz, Vinicius De Rocha, Ana Maria Carpes, François Vernadat, and Xavier Crégut. A Model-Driven Engineering Approach to Formal Verification of PLC programs (regular paper). In *Emerging Technologies and Factory Automation (ETFA), Toulouse, France*, pages 1–8. IEEE, septembre 2011.

[FdQdS+11] Jean-Marie Farines, Max H. de Queiroz, Mateus F. de Souza, Ana Maria M. Carpes, and Francçois Vernadat. Modeling and Verification of Plc Programs by using Fiacre Tool Chain. *TOPCASED DAYS 2011*, 2011.

[FGC+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *Embedded Real Time Software (ERTS)*, Toulouse, France, January 2006.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) - Special issue on POPL 2005*, 29(3), May 2007.

[Fra11] Ulrich Frank. Some guidelines for the conception of domain-specific modelling languages. In Markus Nüttgens, Oliver Thomas, and Barbara Weber, editors, *Enterprise Modelling and Information Systems Architectures: Proceedings of the 4^{th} International Workshop on Enterprise Modelling and Information Systems Architectures, EMISA 2011, Hamburg, Germany, September 22-23, 2011*, pages 93–106, 2011.

[GaG07] Ismênia Galvão and Arda Goknil. Survey of traceability approaches in model-driven engineering. In *11^{th} IEEE International Enterprise Distributed Object Computing Conference EDOC 2007, 15-19 October 2007, Annapolis, Maryland, USA*, pages 313–326, 2007.

[GCKK06] Heather Goldsby, Betty H. C. Cheng, Sascha Konrad, and Stephane Kamdoum. A Visualization Framework for the Modeling and Formal Analysis of High Assurance Systems. In *Proceedings of the 9^{th} international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 707–721, Berlin, Heidelberg, 2006. Springer-Verlag.

[GdLMD09] Esther Guerra, Juan de Lara, Alessio Malizia, and Paloma Díaz. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology*, 51(4):769–784, 2009.

[Ge14] Ning Ge. *Property Driven Verification Framework: Application to Real-Time Property for UML-MARTE Software Designs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, mai 2014.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 372–387, 2011.

[GRS10] A. Gargantini, E. Riccobene, and P. Scandurra. Combining Formal Methods and MDE Techniques for Model-driven System Design and Analysis. *International journal on advances in software*, 3(1,2):1 – 18, 2010.

[HBRV10] Ábel Hegedüs, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of Simulation Traces with Change-Driven Model Transformations. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 145–155, Washington, DC, USA, 2010. IEEE Computer Society.

[HHI+10] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing Graph Transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 205–216, New York, NY, USA, 2010. ACM.

[HHI+11] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations (short paper). In 26th *IEEE/ACM International Conference On Automated Software Engineering (ASE'11)*, pages 480–483. IEEE, 2011.

[HHI+13] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. *Progress in Informatics*, (10):131–148, March 2013. `http://dx.doi.org/10.2201/NiiPi.2013.10.7`.

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, October 2004.

[HSST11] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Dagstuhl seminar on bidirectional transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31 – 39,

2008. Special Issue on Second issue of experimental software and toolkits (EST).

[JB06] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'06, pages 171–185, Berlin, Heidelberg, 2006. Springer-Verlag.

[JCV12] Jean-Marc Jézéquel, Benoît Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique*. Références sciences. Ellipses, February 2012.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag.

[Jou05] Frédéric Jouault. Loosely coupled traceability for ATL. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, 2005.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A Micromodularity Mechanism. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 62–73, New York, NY, USA, 2001. ACM.

[KKP+09] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. Design Guidelines for Domain Specific Languages. In Matti Rossi, Jonathan Sprinkle, Jeff Gray, and Juha-Pekka Tolvanen, editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 7–13, 2009.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 215–229. Springer-Verlag, Berlin, Heidelberg, 2006.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. Polack. The Epsilon Transformation Language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60. Springer-Verlag, Berlin, Heidelberg, 2008.

[KSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

[Lec09] Thierry Lecomte. Applying a formal method in industry: A 15-year trajectory. In Marìa Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 26–34. Springer Berlin Heidelberg, 2009.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer International Publishing, 2014.

[MF76] Philip M. Merlin and David J. Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036 – 1043, sep 1976.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the* 8^{th} *International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278, Berlin, Heidelberg, 2005. Springer-Verlag.

[Min68] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968.

[MP10] Janne Merilinna and Juha Pärssinen. Verification and validation in the context of domain-specific modelling. In *Proceedings of the* 10^{th} *Workshop on Domain-Specific Modeling*, DSM '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[MWVD13] Bart Meyers, Manuel Wimmer, Hans Vangheluwe, and Joachim Denil. Towards Domain-specific Property Languages: The ProMoBox Approach. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, DSM '13, pages 39–44, New York, NY, USA, 2013. ACM.

[OMG03a] Object Management Group. *Model Driven Architecture (MDA) Guide, v1.0.1*, June 2003.

[OMG03b] OMG. *Object Constraint Language (OCL) 2.0*, 2003.

[OMG06] OMG. *Meta Object Facility (MOF) 2.0 Core*, 2006.

[OMG07a] OMG. *Software Process Engineering Metamodel (SPEM) 2.0*, March 2007.

[OMG07b] OMG. *Unified Modeling Language (UML) 2.1.2*, 2007.

[OMG10] OMG. UML 2.3 Superstructure, 2010.

[OMG11a] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[OMG11b] OMG. OMG MOF 2 XMI Mapping Specification, Version 2.4.1, August 2011.

[OMG12] OMG. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012.

[OO+12] Ileana Ober, Iulian Ober, et al. Seeing errors: model driven simulation trace visualization. In *Model Driven Engineering Languages and Systems*, pages 480–496. Springer, 2012.

[Pan07] Marc Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Applications & SystEms Design. In *Model-Driven Development Tool Implementers Forum (MDD-TIF), Zurich, 24/06/2007*, page (on line), http://www.dsmforum.org/events/MDD-TIF07/, juin 2007. Domain Specific Modelling Forum (DSMF).

[Pet81] Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115 – 116, 1981.

[PIM09] Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *IEEE Trans. Softw. Eng.*, 35(3):325–346, May 2009.

[RKK08] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. The ADAPT tool: From AADL architectural models to stochastic petri nets through model transformation. *CoRR*, abs/0809.4108, 2008.

[RL12] Vlad Rusu and Dorel Lucanu. A $\mathbb{K}$-based formal framework for domain-specific modelling languages. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 214–231. Springer Berlin Heidelberg, 2012.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[Rus11] Vlad Rusu. Embedding Domain-specific Modelling Languages in Maude Specifications. *SIGSOFT Software Engineering Notes*, 36(1):1–8, January 2011.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer Berlin Heidelberg, 2008.

[Spi01] Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.

[Ste08] Perdita Stevens. Generative and Transformational Techniques in Software Engineering II. chapter A Landscape of Bidirectional Model Transformations, pages 408–424. Springer-Verlag, Berlin, Heidelberg, 2008.

[VB07] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Sci. Comput. Program.*, 68(3):187–207, October 2007.

[VDW12] Willem Visser, Matthew B. Dwyer, and Michael Whalen. The Hidden Models of Model Checking. *Software & Systems Modeling*, 11(4):541–555, October 2012.

[Wil10] Edward D. Willink. Re-engineering Eclipse MDT/OCL for Xtext. *ECEASST*, 36, 2010.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[WKH13] John Wilson-Kanamori and Soichiro Hidaka. A Bidirectional Collaboration Framework for Bio-Model Development. In 2^{nd} *International Workshop on Bidirectional Transformations (BX 2013)*, 2013.

[WL03] Alan Wassyng and Mark Lawford. Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 133–153. Springer Berlin Heidelberg, 2003.

[YLH+12] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. blinkit: Maintaining Invariant Traceability through Bidirectional Transformations. In *ICSE 2012*, pages 540–550, Zurich, Switzerland, June 2012.

[YMH+06] Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. Pattern based property specification and verification for service composition. In *Proceedings of the 7th International Conference on Web Information Systems*, WISE'06, pages 156–168. Springer-Verlag, Berlin, Heidelberg, 2006.

[ZCP12] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for dsml users: A process modeling case study. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Volume Part II*, ISoLA'12, pages 329–343, Berlin, Heidelberg, 2012. Springer-Verlag.

[ZCP13a] Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal verification integration approach for dsml. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 336–351. Springer Berlin Heidelberg, 2013.

[ZCP13b] Faiez Zalila, Xavier Crégut, and Marc Pantel. A transformation-driven approach to automate feedback verification results. In Alfredo Cuzzocrea and Sofian Maabout, editors, *Model and Data Engineering*, volume 8216 of *Lecture Notes in Computer Science*, pages 266–277. Springer Berlin Heidelberg, 2013.

[ZCP14] Faiez Zalila, Xavier Crégut, and Marc Pantel. A user-oriented approach to integrate formal verification activity for DSML (regular paper). In *International Conference on Embedded Real Time Software and Systems (ERTS2), Toulouse, 05/02/2014-07/02/2014*. SIA/3AF/SEE, février 2014.

[ZG02] Paul Ziemann and Martin Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, volume TUM-I0208, pages 53–62, September 2002.

List of Figures